

UNIVERSITEIT ANTWERPEN

---

# A Foundation for Inconsistency Management in Model-Based Systems Engineering

---

*Een Onderbouw voor Inconsistentiebeheer  
in Modelgebaseerd Systeemontwerp*

*Auteur:*  
István DÁVID

*Promotor:*  
Prof. Dr. Hans VANGHELUWE



*Proefschrift ingediend tot het behalen van de graad van  
Doctor in de Wetenschappen: Informatica*

*To my Father.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Inconsistencies in MBSE . . . . .	2
1.2	Managing <i>inconsistency</i> rather than consistency . . . . .	3
1.3	Research questions . . . . .	3
1.4	Contributions . . . . .	4
1.5	Structure . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Model-based system engineering . . . . .	7
2.1.1	Requirements . . . . .	9
2.1.2	Virtual product . . . . .	10
2.1.3	Models . . . . .	11
2.1.4	Properties and attributes . . . . .	13
2.1.5	Viewpoints, views . . . . .	14
2.1.6	Typical scenarios in MVM . . . . .	16
2.2	Processes . . . . .	17
2.2.1	Modeling . . . . .	19
2.2.2	Enactment . . . . .	19
<b>3</b>	<b>State of the art</b>	<b>21</b>
3.1	(In)consistency management . . . . .	21
3.1.1	Selection method . . . . .	22
3.1.2	What are inconsistencies? . . . . .	25
3.1.3	The origin of inconsistencies . . . . .	26
3.1.4	Managing inconsistencies . . . . .	26
3.1.5	Features and patterns of inconsistency management techniques . . . . .	27
3.1.6	Summary of the state-of-the-art . . . . .	31
3.1.7	Characterization patterns . . . . .	31
3.1.8	Detection patterns . . . . .	37
3.1.9	Resolution patterns . . . . .	40
3.1.10	Optional and auxiliary activities . . . . .	40
3.1.11	Consistency in other contexts . . . . .	42
3.1.12	Conclusions . . . . .	43
3.2	Process engineering . . . . .	44
3.2.1	Process modeling formalisms . . . . .	45
3.2.2	Performance analysis techniques . . . . .	46

<b>4 Correctness and consistency</b>	<b>49</b>
4.1 Correctness . . . . .	50
4.2 The repercussions of ensuring correctness . . . . .	50
4.3 A heuristic for eventual correctness . . . . .	51
4.3.1 Consistency . . . . .	52
4.3.2 The relationship between (in)correctness and (in)consistency . . . . .	52
4.4 The need for explicitly modeled processes . . . . .	53
4.5 System properties for reasoning over consistency . . . . .	54
4.5.1 Architectural decomposition . . . . .	55
4.5.2 View decomposition . . . . .	56
4.5.3 Refinement-abstraction . . . . .	57
4.5.4 Abstraction-refinement . . . . .	58
4.5.5 Elementary engineering operation . . . . .	59
<b>5 Process-oriented inconsistency management</b>	<b>61</b>
5.1 A formalism for modeling engineering processes . . . . .	63
5.1.1 Typed processes in the classic FTG+PM . . . . .	64
5.1.2 Attributes, properties, constraints . . . . .	67
5.1.3 Resources . . . . .	72
5.1.4 Cost models . . . . .	75
5.1.5 ISO/IEC/IEEE 42010:2011 compliant viewpoints . . . . .	77
5.2 Off-line inconsistency management . . . . .	78
5.2.1 Patterns of inconsistency . . . . .	78
5.2.2 Patterns of inconsistency management . . . . .	79
5.2.3 Solving the process optimization problem by multi-objective process-space exploration . . . . .	81
5.3 Process enactment . . . . .	91
5.3.1 Architecture . . . . .	91
5.3.2 Execution semantics . . . . .	92
5.3.3 Transformation rules . . . . .	93
5.3.4 Implementation . . . . .	94
5.4 On-line inconsistency management . . . . .	95
5.4.1 Modeling support . . . . .	97
5.4.2 Modeling the running example . . . . .	98
5.4.3 Early detection of inconsistencies . . . . .	103
5.4.4 Discussion . . . . .	105
5.5 Translating process models to DEVS . . . . .	106
5.5.1 Translating processes to DEVS . . . . .	109
5.5.2 Calibration of the process models . . . . .	113
5.5.3 Modeling resource constraints . . . . .	115
5.5.4 Performance evaluation . . . . .	118
5.5.5 Conclusions . . . . .	119
5.6 Modeling service interactions . . . . .	119
5.6.1 The Statecharts + Class Diagrams (SCCD) formalism . . . . .	120
5.6.2 Motivating example . . . . .	120
5.6.3 Modeling activities using SCCD . . . . .	122
5.6.4 Mapping Processes to SCCD . . . . .	123
5.6.5 Conclusions . . . . .	128

5.7	Inconsistency tolerance . . . . .	129
5.7.1	Motivating examples . . . . .	129
5.7.2	System properties and consistency statements . . . . .	130
5.7.3	Other applicable cases . . . . .	132
5.7.4	A quantitative model for assessing inconsistencies . . . . .	132
5.7.5	Towards tolerance of inconsistencies . . . . .	134
5.7.6	Discussion . . . . .	138
5.7.7	Conclusions . . . . .	140
<b>6</b>	<b>Case study</b>	<b>141</b>
6.1	Introduction . . . . .	141
6.2	Modeled reproduction of the hard-coded process . . . . .	144
6.3	Modeling the case using our approach . . . . .	147
6.3.1	Process modeling . . . . .	147
6.3.2	Attributes, properties, constraints . . . . .	152
6.3.3	Properties . . . . .	154
6.3.4	Resources . . . . .	155
6.3.5	Costs . . . . .	155
6.3.6	Off-line inconsistency management . . . . .	156
6.3.7	On-line inconsistency management . . . . .	158
<b>7</b>	<b>Prototype tooling</b>	<b>161</b>
7.1	Features . . . . .	161
7.2	Architecture . . . . .	161
7.3	Modeling of engineering processes . . . . .	163
7.3.1	Language specification . . . . .	164
7.3.2	Eight important modeling patterns . . . . .	166
7.3.3	Specifying executable scripts for automated activities . . . . .	170
7.3.4	Code generation . . . . .	170
7.3.5	Validation . . . . .	170
7.4	Off-line inconsistency management . . . . .	170
7.5	Enactment of engineering processes . . . . .	170
7.6	On-line inconsistency manager . . . . .	171
7.7	Service integration . . . . .	172
7.8	Reflection and future directions . . . . .	172
<b>8</b>	<b>Conclusions</b>	<b>173</b>
8.1	Evaluation of the research questions . . . . .	176
8.2	Future research directions . . . . .	179
<b>Appendices</b>		<b>181</b>
<b>A</b>	<b>Metamodels</b>	<b>183</b>
A.1	Overview . . . . .	184
A.2	FTG+PM . . . . .	185
A.3	Properties . . . . .	186
A.4	Resources . . . . .	187
A.5	Costs . . . . .	188
A.6	Viewpoints . . . . .	189

A.7 Enactment . . . . .	190
<b>B Model queries and transformations</b>	<b>191</b>
<b>C DEVS library for processes</b>	<b>201</b>
<b>D Artifacts of the AGV case study</b>	<b>209</b>

# Acknowledgements

Throughout my life, I have been always striving for greatness in my work. That's how I was raised and trained. By my parents, my teachers, my co-workers, my superiors. Through the past four years I have learned the value of credibility, accountability, persistence and loyalty. Qualities that define us not just professionally, but personally as well. Strangely enough, working towards one of the highest professional degrees also made me understand how to be a better person. That is not to say, I didn't grow as a professional. No, that is to say, that the people around me thought me important lessons about greater things. And I will be always thankful for that.

First of all, I would like to thank my supervisor, Hans. You trusted me and gave me a chance four years ago. Quite frankly, I still don't get what you saw in me during our limited conversations. And quite frankly, I never felt I can live up to your standards. But that's probably because you are that one-in-a-million type of a person one shouldn't compare himself to, and one should feel exceptionally lucky to work with. Being a great person you are, you never made me feel secondary or inferior. You treated me like you treat everyone: with respect and friendliness. You took your time to teach me lessons and help me grow. You were there during the toughest periods of my research; and you were there during the toughest period of my personal life. You were a true *doktorvater*, in every meaning of the word.

So thank you, Hans, for helping me become who I am today. And even though we didn't achieve world dominance while I was working with you, I can promise that I'll carry the knowledge and experiences I got from you and I will spread the philosophy we stand for.

I would like to thank you, dear colleagues and friends, who stood by me during these trying years. Yentl, Joachim, Simon, Ken, Bart and Claudio, your presence created the most inspiring professional environment I've ever got the chance to work within. My thanks goes to the larger group of AnSyMo as well, especially Ali, Tim, Naomi.

I would like to thank the patience of the people I shared an office with, Valérie and Fons. Your friendliness and the wholesome conversations always meant a great deal to me.

I would like to express my gratitude to the members of my PhD committee and jury: Antonio Cicchetti, Joachim Denil, Serge Demeyer, Klaas Gadeyne, Tom Mens and Dirk Janssens. Your feedback, remarks and pointers were essential to arriving to the point of writing and submitting this thesis.

I would like to thank the Flanders Make and their co-workers for the three years of

joint experience on the MBSE4Mechatronics project. Klaas, Maarten and Kristof, you are absolutely remarkable professionals who thought me the value of science outside the realm of academia.

Throughout my PhD years I got the chance to *walk the path to enlightenment* with the greatest of our community. Thank you for accepting me as one of you. Thank you for teaching me new things. Thank you for sharing your professional ideas. And thank you, Pieter, for sharing your personal experiences during those unforgettable 36 hours of December 2016. I would like to thank the people who gave me the rock solid foundations I was building on top of in the past four years: István, Ákos and Laci.

Last but not least, I would like to thank everything to a special person who stood by me as long as she could. Thank you, Barbara, for being my rock during these stormy and uncertain years. Thank you for being the companion throughout this long, tiring, trying and exhausting journey. Even though you (jokingly) view my work as “drawing boxes and arrows”, this thesis would have never been started, let alone completed, without you. Thank you.

*Istvan*

# Abstract

The complexity of current engineered systems has increased drastically over the last decades. A pertinent example are today's mechatronic and cyber-physical systems (CPS). These are characterized by heterogeneity, namely the complex interplay between physical, software, and network components. With the advent of such systems, and nowadays' surging Industry 4.0 initiative, engineers are facing challenges of an unprecedented magnitude.

Tackling the complexity of heterogeneous systems in an effective and efficient manner is not a trivial task. Model-based and model-driven techniques propose modeling the system before it gets realized (implemented, manufactured, etc). The aim of such a modeling activity is to have an abstracted representation of the system, and that, with different intents. Models, at the very least, can be used as a documentation of the system, either in a descriptive or in a prescriptive way. But formal models also enable the analysis of the future product, typically through simulation, validation or verification techniques. By lifting such analysis activities to the early phases of the engineering endeavor, the engineering of the product becomes more feasible in terms of the actual realizability, the quality of the product and the associated costs.

Approaches such as Model-based system engineering (MBSE) and Model-driven engineering (MDE) treat models as first-class citizens. That is, modeling is not an optional element of engineering, but a foundational concept that the engineering activities are based on or driven by. Domain-specific modeling (DMS), specifically, advocates narrowing the scope of models to a specific problem of a specific domain. Domain-specific modeling languages (DS(M)L) are explicitly tailored for enabling an efficient work of the domain expert who creates the models. Multi-paradigm modeling (MPM) advocates embracing the fact that in efficient MBSE/MDE settings, multiple DSLs are present at a time. The idea behind MPM is to modeling everything explicitly, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction. Intuitively, every engineering setting follows MPM principles, albeit with varying adherence to the previous principles.

Employing an MPM approach to engineer a heterogeneous systems will help tackling the intrinsic complexity of the given system. Employing an MPM approach, however, will introduce a different type of complexity, that stems from the very diverse set of modeling formalisms used. This complexity, however, is more manageable than the intrinsic complexity of the engineered system. Often, the parallel of accidental and essential complexity is used to discuss this phenomenon. Engineers, ideally, should spend their time addressing the essential complexity of the problem; but that is only feasible if the accidental complexity is being taken care of by smart techniques, tools and machinery, realized by the

modeling tools they use.

One important challenge in MPM is the presence of inconsistencies between the various domain-specific models. The phenomenon of model inconsistency is especially concerning when modeling formalisms of disparate semantic domains are employed for the same particular engineering endeavor. Consider identifying an inconsistency between a Simulink model and a continuous fluid dynamics model.

Inconsistency management has been a well-studied topic in the domains of software engineering, mechatronic design and cyber-physical systems, due to the typically MPM approach to the system design. Unfortunately, most of the state-of-the-art techniques focus on inconsistencies which manifest on the syntactic level of modeling languages. This is merely a symptomatic treatment of the problem. As pointed out in the recent years, it is the overlaps in the semantic domain of models that are responsible for the eventual inconsistency. Given that nowadays' engineering-modeling methodologies are not explicit about the semantics of the modeling formalisms, addressing inconsistencies on the semantic level is extremely challenging.

In this thesis we summarize the results of a four year scientific endeavour, during which we have been investigating the problem of managing semantic inconsistencies, motivated by the engineering of complex heterogeneous systems.

Our contributions consists of

- a systematic literature overview on the topic of inconsistency management;
- a conceptual approach to managing inconsistencies in heterogeneous engineering settings; and
- a state-of-the-practice tooling to support our methodology with.

Instead of removing inconsistencies, our approach aims at managing inconsistencies. That is, various strategies of coping with inconsistencies can be evaluated and applied for a specific inconsistency instance. In some cases, preventive techniques (such as contract-based design) are the efficient ones; while in other cases, the presence of inconsistencies can be tolerated and fixed later. Such a reasoning requires quantified metrics as of what impact a given inconsistency management technique will have on the overall engineering endeavor.

To this end, and in the spirit of MPM, we propose modeling explicitly everything that is related to the problem of inconsistencies during the engineering endeavour, most notably: the flow of the engineering endeavor itself, what we, from now on, will call the engineering process.

Our approach consists of multiple contributions to the state-of-the-art. These are the following.

- Revising the definition of model (in)consistency, in order to support the management of semantic inconsistencies.
- A process modeling formalism, serving as the foundation for the rest of the contributions.

- Off-line management of inconsistencies by the means of rule-based multi-objective process-space exploration.
- Enactment of the optimized process by the means of explicit model transformations.
- On-line management of inconsistencies by monitoring the process under enactment.
- DEVS-based simulation of processes for evaluating the performance of the process candidates in the off-line management phase.
- External service integration in processes under enactment, by explicit transformations over SCCD models.

We demonstrate the feasibility of our contributions through various examples and case studies.



# **Nederlandstalige Samenvatting**



# Publications

The following peer-reviewed publications that I co-authored were included (partially) in this thesis:

1. DÁVID, I., VAN TENDELOO, Y. AND VANGHELUWE, H. Translating Engineering Workflow Models to DEVS for Performance Evaluation. *In Proceedings of the Winter Simulation Conference*. (2018)

*Istvan and Hans came up with the ideas; Istvan and Yentl elaborated on the ideas. Yentl implemented the approach in the Modelverse. Istvan and Yentl wrote the paper; Hans reviewed the paper.*

2. DÁVID, I., DENIL, J. AND VANGHELUWE, H. Process-oriented Inconsistency Management in Collaborative Systems Modeling. *16th annual Industrial Simulation Conference (ISC)*. (2018)

*Istvan and Hans came up with the ideas; Istvan and Joachim elaborated on the ideas. Istvan implemented the approach. Istvan wrote the paper; Hans reviewed the paper.*

3. VAN MIERLO, S., VAN TENDELOO, Y., DÁVID, I., MEYERS, B., GEBREMICHAEL, A., AND VANGHELUWE, H., A Multi-Paradigm Approach for Modelling Service Interactions in Model-Driven Engineering Processes. *In Model-driven Approaches for Simulation Engineering Symposium (MOD4SIM) – Spring Simulation Multi-Conference*. (2018)

*Bart and Hans came up with the ideas; Yentl and Addis elaborated on the ideas. Addis and Yentl implemented the approach. Simon, Yentl, and Istvan wrote the paper; Hans and Addis reviewed the paper.*

4. DÁVID, I., MEYERS, B., VANHERPEN, K., VAN TENDELOO, Y., BERX, K., AND VANGHELUWE, H., Modeling and Enactment Support for Early Detection of Inconsistencies in Engineering Processes. *2nd International Workshop on Collaborative Modelling in MDE*. (2017)

*Istvan and Hans came up with the ideas; Istvan, Bart, Ken and Yentl elaborated on the ideas. Istvan implemented the approach. Istvan wrote the paper; Bart, Ken, Yentl and Hans reviewed the paper.*

5. DÁVID, I., SYRIANI, E., VERBRUGGE, C., BUCHS, D., BLOUIN, D., CICCHETTI, A. AND VANHERPEN, K. Towards Inconsistency Tolerance by Quantification of Semantic Inconsistencies. *1st International Workshop on Collaborative Modelling in MDE*. (2016)

*Istvan, Eugene, Clark, Didier, Dominique, Antonio and Ken came up with the idea. Istvan, Eugene, Dominique, Clark and Didier wrote the paper. Ken reviewed the paper.*

6. DÁVID, I., DENIL, J., GADEYNE, K., AND VANGHELUWE, H. Engineering Process Transformation to Manage (In)consistency. *1st International Workshop on Collaborative Modelling in MDE*. (2016)

*Istvan and Hans ideas; Istvan and Joachim elaborated on the idea. Istvan implemented the approach. Istvan wrote the paper. Joachim, Hans and Klaas reviewed the paper.*

7. DÁVID, I. A Multi-Paradigm Modeling Foundation for Collaborative Multi-view Model/System Development. *ACM Student Research Competition (SRC) MoDELS*. (2016)

*Istvan and Hans came up with the ideas. Istvan implemented the approach. Istvan wrote the paper; Hans reviewed the paper.*

8. VANHERPEN, K., DENIL, J., DÁVID, I., DE MEULENAERE, P., MOSTERMAN, P. J., TÖRNGREN, M., QAMAR, A., AND VANGHELUWE, H. Ontological Reasoning for Consistency in the Design of Cyber-Physical Systems. *CPPS 2016 - 1st International Workshop on Cyber-Physical Production Systems*. (2016)

*Ken, Hans and Joachim came up with the ideas. Ken, Paul, Pieter, Martin, Ahsan and Istvan elaborated on the idea. Ken and Istvan wrote the paper. Joachim, Istvan, Paul, Pieter, Martin, Ahsan and Hans reviewed the paper.*

9. DÁVID, I., DENIL, J., AND VANGHELUWE, H. Patterns of inconsistency management in mechatronics - A survey. *Technical report*. (2015)

*Istvan and Joachim did the literature survey. Istvan and Joachim categorized the processed literature. Istvan wrote the paper. Joachim and Hans reviewed the paper.*

10. DÁVID, I., DENIL, J., AND VANGHELUWE, H. Towards Inconsistency Management by Process-Oriented Dependency Modeling. *In MPM 2015 - 9th International Workshop on Multi-Paradigm Modeling*. (2015)

*Istvan and Hans came up with the ideas; Istvan and Joachim elaborated on the ideas. Istvan worked out the manual approach. Istvan wrote the paper; Hans reviewed the paper.*

The following peer-reviewed publications that I co-authored were not included in this thesis:

1. MEYERS, B., DENIL, J., DÁVID, I., AND VANGHELUWE, H. Automated Testing Support for Reactive Domain-Specific Modelling Languages. *In 9th International Conference on Software Language Engineering Conference (SLE)*. (2016)
2. DÁVID, I., RÁTH, I., AND VARRÓ, D. Foundations for Streaming Model Transformations by Complex Event Processing. *International Journal on Software and Systems Modeling* (2016), pp 1–28. DOI: 10.1007/s10270-016-0533-1.
3. BALOGH, L., DÁVID, I., RÁTH, I., VARRÓ, D., AND VÖRÖS, A. Distributed and Heterogeneous Event-based Monitoring in Smart Cyber-Physical Systems (Extended abstract). *MT CPS 2016 - 1st Workshop on Monitoring and Testing of Cyber-Physical Systems*. (2016)
4. BERGMANN, G., DÁVID, I., HEGEDÜS, Á., HORVÁTH, Á., RÁTH, I., UJHELYI, Z., AND VARRÓ, D. VIATRA3: A Reactive Event-driven Model Transformation Platform. *Theory and Practice of Model Transformations (D. Kolovos, M. Wimmer, eds.), vol. 9152 of Lecture Notes in Computer Science. Springer International Publishing*. (2015), pp. 101-110.
5. DÁVID, I., RÁTH, I., AND VARRÓ, D. Streaming model transformations by complex event processing. *In Model-Driven Engineering Languages and Systems (J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, eds.), vol. 8767 of Lecture Notes in Computer Science, Springer International Publishing*. (2014), pp. 68-83. Invited for submission to MODELS14 Special Issue in SoSyM
6. DÁVID, I., AND GÖNCZY, L. Ontology-Supported Design of Domain-Specific Languages: A Complex Event Processing Case Study. *In Advances and Applications in Model-Driven Engineering (Ed.: Dr. Vicente García Díaz, Prof. Dr. Juan Manuel Cueva Lovelle, Dr. Begoña Cristina Pelayo García-Bustelo and Dr. Oscar Sanjuán Martínez), IGI Global*. (2013) Book chapter.
7. DÁVID, I. A model-driven approach for processing complex events. *In CoRR, abs/1204.2203, Student Forum of the Ninth European Dependable Computing Conference - EDCC, Sibiu, Romania*. (2012)
8. DÁVID, I., AND BUZA, K. On the relation of cluster stability and early classifiability of time series. *In 36th Annual Conference of the German Classification Society on Data Analysis, Machine Learning and Knowledge Discovery, Hildesheim, Germany*. (2012), p. 91
9. DÁVID, I. Modellalapú fejlesztési módszer komplex események feldolgozásához (Hungarian only). *In Mesterpróba Tudományos Konferencia, Budapest, Hungary*. (2012), pp. 1-4



# **Overview of activities**

## **Teaching, course work**

- Home assignment supervision and grading for the course “Modelling of Software-Intensive Systems” at University of Antwerp (2015 - 2018).
- Project supervision for the course “Model-driven engineering” at University of Antwerp (2015 - 2018).

## **Scientific activities**

- Conference talk at the 16th annual Industrial Simulation Conference (ISC 2018), Ponta Delgada, Portugal, 2018 May
- Participation at the MPM4CPS COST Action, Riga, 2018 April
- Conference talk at the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Austin, Texas, 2017 October
- Multiple talks at the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS), St Malo, France, 2016 October
- Engineering Process Transformation to Manage (In)consistency in Complex Systems, Tallinn Tech, Tallinn, Estonia, 2016 June
- A Multi-Paradigm Modeling Foundation for Collaborative Multi-view Model/System Development, Université de Mons, Belgium, 2016 June
- Complex event processing with the VIATRA platform, IncQuery Academia, Budapest, Hungary, 2016 May
- Invited participant at the 16th Computer Automated Multi-Paradigm Modelling workshop, Bellairs Research Center, Barbados, 2016 May
- Participant at the 6th International Summer School on Domain-Specific Modeling (DSM-TP), Antwerp, Belgium, 2015 August
- Participant at the EPFL / ETH Summer School on DSL Design and Implementation, École polytechnique fédérale de Lausanne, Switzerland, 2015 July

- Conference talk at the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Valencia, Spain, 2014 October

## Reviewing

- ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)
- Federated Conference on Computer Science and Information Systems (FedCSIS)
- Workshop on Model-Driven Requirements Engineering (MoDRE)

# List of Figures

1.1 Domains typically involved in a mechatronic design process. . . . .	2
1.2 Overview of the approach, depicted as an FTG+PM. . . . .	5
1.3 Overview of the <i>Off-line inconsistency management</i> compound activity . .	6
2.1 General overview of the engineering process. . . . .	8
2.2 Model-based engineering. . . . .	8
2.3 The typical workflow of requirements engineering. . . . .	9
2.4 Modeling and simulation concepts . . . . .	12
2.5 Model-based systems analysis. . . . .	13
2.6 The Core of Architecture Description in ISO/IEC/IEEE 42010:2011 . . .	15
2.7 Scenarios in multi-view modeling. . . . .	16
3.1 High-level workflow of searching, selecting and processing. . . . .	22
3.2 Detailed selection process. . . . .	24
3.3 Feature model of the inconsistency management techniques. . . . .	27
3.4 Conceptual overview of inconsistency management styles and activities. .	29
3.5 Extended view on inconsistency management styles and techniques. . .	44
4.1 The relationship between properties and designs. . . . .	49
4.2 Correctness often contradicts efficiency. . . . .	51
4.3 Correctness is approximated by consistency; efficiency is measured in terms of the process. . . . .	54
4.4 The five types of engineering activities. . . . .	55
4.5 The sub-process of <i>Architectural decomposition</i> and <i>System integration</i> . .	56
4.6 The sub-process of <i>View decomposition</i> and <i>View merge</i> . . . . .	57
4.7 The sub-process of <i>Refinement-Abstraction</i> . . . . .	58
4.8 The sub-process of <i>Abstraction-Refinement</i> . . . . .	59
4.9 The sub-process of the <i>Elementary engineering operation</i> . . . . .	60
5.1 Front and top view of the conceptual design of the AGV. . . . .	62
5.2 Running example. . . . .	62
5.3 Relationships between processes, formalisms and properties. . . . .	63
5.4 Excerpt of the FTG+PM part of the modeling formalism. . . . .	65
5.5 The PM of the running example. . . . .	66
5.6 The FTG of the running example. . . . .	66
5.7 The FTG+PM of the running example. . . . .	67

5.8	Influence relationship between two attributes. . . . .	68
5.9	Causal and acausal notation of a relationship. . . . .	69
5.10	The running example, extended with attributes, constraints and intents . . . . .	72
5.11	Excerpt of the resource modeling part of the formalism. . . . .	73
5.12	Resource types and resources of the running example. . . . .	74
5.13	Resource demands of the running example. . . . .	75
5.14	Resource allocations of the running example. . . . .	75
5.15	Excerpt of the cost modeling part of the formalism. . . . .	76
5.16	Associated costs of the running example. . . . .	77
5.17	Detailed overview of the search process. . . . .	81
5.18	Structural overview of the DSE approach. . . . .	82
5.19	Architectural overview of the design space exploration component . . . . .	83
5.20	Constraints and objectives of the DSE. . . . .	84
5.21	Transformation rules of the DSE. . . . .	84
5.22	Design space exploration mechanism. . . . .	86
5.23	The PM of the DSE mechanism. . . . .	87
5.24	A process model and its quantitative SimEvents simulation model. . . . .	89
5.25	Two managed alternatives of the process in Figure 5.10. . . . .	90
5.26	Architectural overview of the enactment engine. . . . .	92
5.27	Metamodel for the enactment. . . . .	93
5.28	Excerpt from the metamodel for early detection . . . . .	97
5.29	Attributes and constraints. . . . .	98
5.30	A capability and its constraint in the FTG. . . . .	99
5.31	Excerpt from the example: the property <i>validMass</i> . . . . .	100
5.32	The process model with the capabilities and the attributes . . . . .	102
5.33	The process of selecting the optimal hardware component. . . . .	108
5.34	FTG+PM model of our approach. . . . .	109
5.35	Transformation rule for control flow links. . . . .	113
5.36	The augmented process. . . . .	114
5.37	Results of the simulations. . . . .	118
5.38	The process model of the example. . . . .	121
5.39	Protocol for communicating with an external service . . . . .	123
5.40	Optimize rules. . . . .	126
5.41	Orchestrator rule. . . . .	126
5.42	Activity rule. . . . .	126
5.43	Fork rule. . . . .	127
5.44	Join rule. . . . .	127
5.45	Decision rule. . . . .	128
5.46	Process model from Figure 5.38 mapped to SCCD. . . . .	128
5.47	Overview of the Automated Guided Vehicle example. . . . .	130
5.48	$\beta$ -bounded consistency of two views. . . . .	135
5.49	Exact consistency of two views with $\beta = 0$ . . . . .	136
5.50	Eventual consistency of two views with $\beta = 0$ . . . . .	137
5.51	Repetitive consistency of two views. . . . .	137
5.52	Regular consistency of two views. . . . .	138
6.1	The automated guided vehicle of the case study. . . . .	141
6.2	Front and top view of the conceptual design of the AGV. . . . .	142

6.3	The FTG+PM of the case study. . . . .	145
6.4	Typed activity with the Matlab script <i>motorSelection.m</i> providing the execution semantics. . . . .	146
6.5	The built-in functionality for generating Matlab scripts shown in the menu. . . . .	147
6.6	The PM of the case. . . . .	149
6.7	The FTG of the case. . . . .	150
6.8	The augmented PM of the case. . . . .	153
6.9	The resource model of the case. . . . .	155
6.10	Invoking the optimization mechanism on the PM. . . . .	156
6.11	The optimized PM of the case. . . . .	157
6.12	Alias table for some of the attributes. . . . .	159
6.13	Invoking the process enactment mechanism on the PM. . . . .	159
6.14	The enacted PM with the relevant attributes and capabilities of the on-line inconsistency management phase. . . . .	160
7.1	General architecture of the tool. . . . .	162
7.2	Various graphical components of the modeling UI. . . . .	163
7.3	Representation types for the <i>Process Model</i> . . . . .	164
7.4	Representation types for the PM. . . . .	165
7.5	Elements for modeling the PM. . . . .	165
7.6	Elements for modeling the FTG. . . . .	165
7.7	Elements for modeling the typing in the FTG+PM. . . . .	166
7.8	Elements for modeling the properties. . . . .	166
7.9	Additional elements for modeling the properties. . . . .	166
7.10	Additional elements for modeling the resources. . . . .	166
7.11	Intents. . . . .	167
7.12	Exogenous model transformation. . . . .	167
7.13	Endogenous model transformation. . . . .	168
7.14	Endogenous in-place model transformation. . . . .	168
7.15	Property refinement. . . . .	168
7.16	Performance value derived through an L3 relationship. . . . .	169
7.17	Explicit simulation of a performance value. . . . .	169
7.18	Property-capability typing. . . . .	169
A.1	Top-level overview of the formalism. . . . .	184
A.2	The FTG+PM modeling part of the formalism. . . . .	185
A.3	The property modeling part of the formalism. . . . .	186
A.4	The resources modeling part of the formalism. . . . .	187
A.5	The cost modeling part of the formalism. . . . .	188
A.6	The viewpoint modeling part of the formalism. . . . .	189
A.7	The enactment modeling part of the formalism. . . . .	190



# List of Tables

3.1	Features of inconsistency management and their interpretation. . . . .	28
3.2	Inconsistency management dimensions and their patterns. . . . .	28
3.3	Summary on the features of state inconsistency management techniques. .	30
3.4	Inconsistency management patterns in the state of the art. . . . .	39
3.5	Summary of the related work on processes. . . . .	46
4.1	Consistency does not imply correctness. . . . .	52
5.1	Example consistency properties. . . . .	131
6.1	Execution times of the activities. . . . .	156



# List of Listings

5.1	The read-modify inconsistency pattern. . . . .	83
5.2	Management alternatives of the read-modify inconsistency pattern. . . . .	85



# List of abbreviations

- AGV** Automated guided vehicle  
**CBD** Contract-based design  
**DEVS** Discrete event system specification  
**DSE** Design space exploration  
**EMF** Eclipse Modeling Framework  
**FTG** Formalism-transformation graph  
**FTG+PM** Formalism-transformation graph + Process model  
**ICM** Inconsistency management  
**M2M** Model to model (transformation)  
**M2T** Model to text (transformation)  
**MBSE** Model-based system engineering  
**MT** Model transformation  
**PDEVS** Parallel DEVS  
**PM** Process model  
**RCPSP** Resource constrained project scheduling problem  
**RSM** Response surface methodology  
**SCCD** Statecharts + Class Diagrams  
**TGG** Triple-graph grammar  
**VP** Virtual product



# Chapter 1

## Introduction

The complexity of current engineered systems has increased drastically over the last decades. A pertinent example are today’s mechatronic and cyber-physical systems (CPS). These are characterized by heterogeneity, namely the complex interplay between physical, software, and network components [148].

Due to their complexity, these systems are no longer engineered by a single individual, but rather by the collaboration of experts. Such collaborative endeavors involve stakeholders from different domains, who bring their point of view on the system to be built, resulting in typical settings of multi-view and multi-paradigm modeling (MPM) [133], which proposes to tackle complexity by modeling and relating all aspects of the system – including development processes – explicitly, using the most appropriate formalisms, at the most appropriate levels of abstraction.

Collaborative modeling scenarios are vulnerable to model inconsistencies. This is a consequence of the multiple views on the same virtual product that give rise to outdated and incorrect data. Overlaps in the semantic domain of models have been identified as the primary reason of model inconsistencies by many authors [148, 91, 208]. That is, properties of different models often turn out to be logically connected or sometimes even (nearly) the same [71, 83]. Such a property can be, for example, the “is safe” property of the designed system, which in turn can be implied by the property of “is stable” of a specific subsystem, meaning that the two properties are connected as they semantically overlap. Involving different engineering domains which typically feature disparate modeling formalisms further aggravates the problem.

Although the problem of inconsistencies is a well-studied area in software engineering [172], state-of-the-art techniques typically focus on syntactic inconsistencies [156, 75, 78]. Since retaining semantic consistency is typically linked to simulation and model checking techniques which can often be resource demanding and time consuming, state-of-the-art techniques fail to efficiently address the consistency issues of semantic properties in the broader system engineering domain.

## 1.1 Inconsistencies in MBSE

Inconsistency management is a well-studied topic in the domains of software engineering, mechatronic design and cyber-physical systems, due to the typically multi-view and often multi-paradigm approach to system design. Persson et al [148] identify consistency between the various views of cyber-physical system design as one of the main challenges in design of such complex systems. This is due to relations between views, with respect to their semantic relations, process and operations which often overlap. Our technique embraces these ideas and addresses the problem of inconsistencies by explicitly modeling semantic properties and relating them to engineering processes.

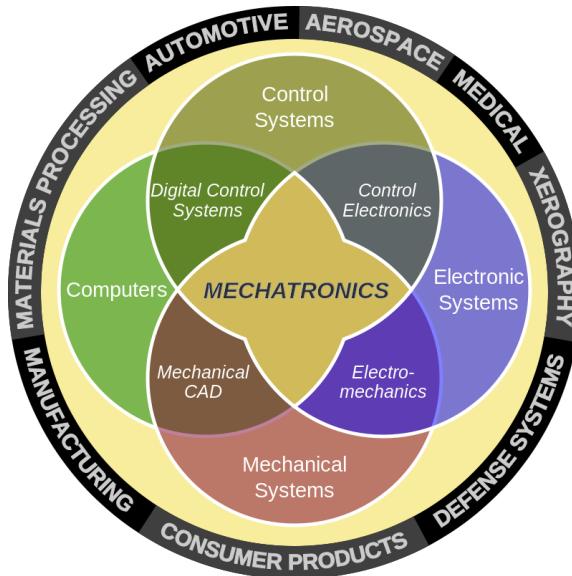


Figure 1.1: Domains typically involved in a mechatronic design process. (Source: <http://www.rpi.edu.>)

One of the key issues in the development of modern mechatronic systems is the strict integration of mechanical, control, electrical and electronic as well as software aspects from the beginning of the earliest design phases on. [88] Due to the conceptual distance between the involved engineering domains, facilitating an efficient mechatronic process is not trivial. The differing vocabularies, techniques, tools and practices make this goal very difficult to achieve. The need for integration of different engineering domains emerges already in early stages of mechatronic processes, as the eventually delivered complex systems are contemplated extensively through model-based approaches. [150]

Modeling various aspects of complex systems needs to be approached using the most appropriate formalisms, levels of abstraction, techniques and tools. Heterogeneous layouts like this inherently suffer from emerging inconsistencies among various models, which can lead to bad design, invalid products and eventually, enormous increase in associated costs and time-to-market.

## 1.2 Managing *inconsistency* rather than consistency

This seems like an unnecessary nitpicky assertion, but thesis is concerned with managing inconsistencies rather than consistencies. It became clear, already around the very onset of this research, that these two terms, and the sort of dual problems behind them, are not to be interchanged.

Managing consistency is traditionally associated with the preventive types of management techniques. One wants to manage consistency with the obvious goal of retaining consistency.

We claim, that inconsistencies are to be embraced as the parts of the engineering endeavour. They obviously must be resolved eventually. But their sole presence at a specific point of the engineering process does not mean a necessary bad (or good, for that matter).

It was Finkelstein [70], who claimed that “*Rather than thinking about removing inconsistency we need to think about managing consistency*”. We firmly stand by this assertion. We also extend this idea, making the following the main message of this thesis.

Rather than managing consistency, we need to think about managing inconsistency.

It is always important to maintain a coherent, or, if you wish, a *consistent* terminology, especially in the scientific literature. From now on, therefore, we will refer to the union *consistency* and *inconsistency* problems/techniques/frameworks/etc, simply by the term *inconsistency* problem/technique/fremework/etc.

## 1.3 Research questions

We formulate a handful of research questions in order to organize our research.

- R1** What are the shortcomings of the state-of-the-art of inconsistency management that hinder the model based engineering of heterogeneous systems?
- R2** What is the relation of model (in)consistency to the (in)correctness of the product?
- R3** What is the most appropriate level of abstraction to approach inconsistency management in model based systems engineering scenarios?
- R4** What formalisms are required to successfully model an engineering endeavor with the intent of identifying inconsistencies across the various models?
- R5** How can be the impacts of applying different inconsistency management patterns quantified?
- R6** Can an inconsistency management technique go beyond being prescriptive and actually enact the inconsistency management techniques chosen for a particular case?

At the end of this thesis, in Section 8.1 we will answer the questions in greater details, linking them to the contributions, presented below in Section 1.4.

## 1.4 Contributions

The contributions of this work are the following.

**Contribution 1: A thorough review of the state of the art,** in order to identify the shortcomings of the currently available inconsistency management techniques. (Chapter 3)

**Contribution 2: Revising the definition of model (in)consistency,** in order to support the management of semantic inconsistencies. (Chapter 4)

**Contribution 3: A process modeling formalism,** serving as the foundation for the rest of the contributions. (Section 5.1)

**Contribution 4: Off-line management of inconsistencies** by the means of rule-based multi-objective process-space exploration. (Section 5.2)

**Contribution 5: Enactment of the optimized process** by the means of explicit model transformations. (Section 5.3)

**Contribution 6: On-line management of inconsistencies** by monitoring the process under enactment. (Section 5.4)

**Contribution 7: DEVS-based simulation of processes** for evaluating the performance of the process candidates in the off-line management phase. (Section 5.5)

**Contribution 8: External service integration in processes** under enactment, by explicit transformations over SCCD models. (Section 5.6)

An additional facet of inconsistency management techniques, **inconsistency tolerance** has been briefly researched in the past four years as well. The results are discussed in Section 5.7. The topic itself proved to be too broad to be thoroughly researched and therefore, we only discuss the concepts and directions toward inconsistency tolerance by quantification of semantic inconsistencies.

The above contributions build up a process for an approach for inconsistency management, depicted in Figure 1.2 and Figure 1.3 by the means of the FTG+PM formalism. [121] Most of the contributions provide automated machinery for carrying out the specific steps of the approach. The only exception, naturally, is the process modeling step, which still has to be carried out manually. (This can be enhanced by various process mining and data mining approaches, but these do not constitute the focus of this research.) The automation of the majority of the steps is achieved by an appropriate tooling, presented later in this thesis.

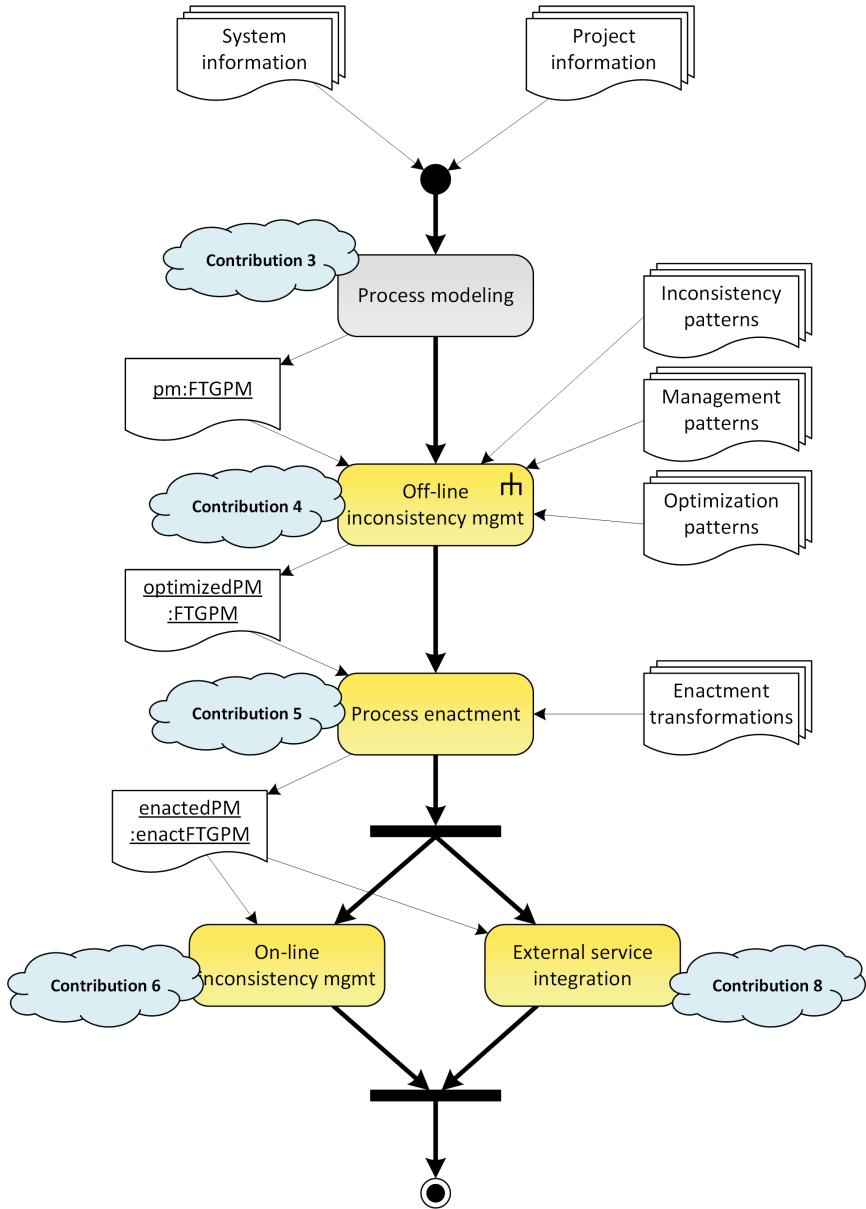


Figure 1.2: Overview of the approach, depicted as an FTG+PM.

## 1.5 Structure

Chapter 2 provides the necessary background information on the techniques and foundations relevant to this thesis. Chapter 3 reviews the state-of-the-art (SOTA) related to this thesis. This entails (i) (in)consistency management, and (ii) process engineering, both constituting a vast body of knowledge. Chapter 4 serves as the foundational theoretical baseline

for this thesis and is based on distilling and extending the state-of-the-art. Chapter 5 presents the foundations for process-oriented inconsistency management with all of its facets. Chapter 6 evaluates the previously presented techniques by applying them to a demonstrate example. Chapter 7 is a reflection on the current tooling supporting the previously presented techniques. It provides an overview on the current state of the tool and directives for future development. Finally, Chapter 8 concludes this thesis and points towards possible directions for future work.

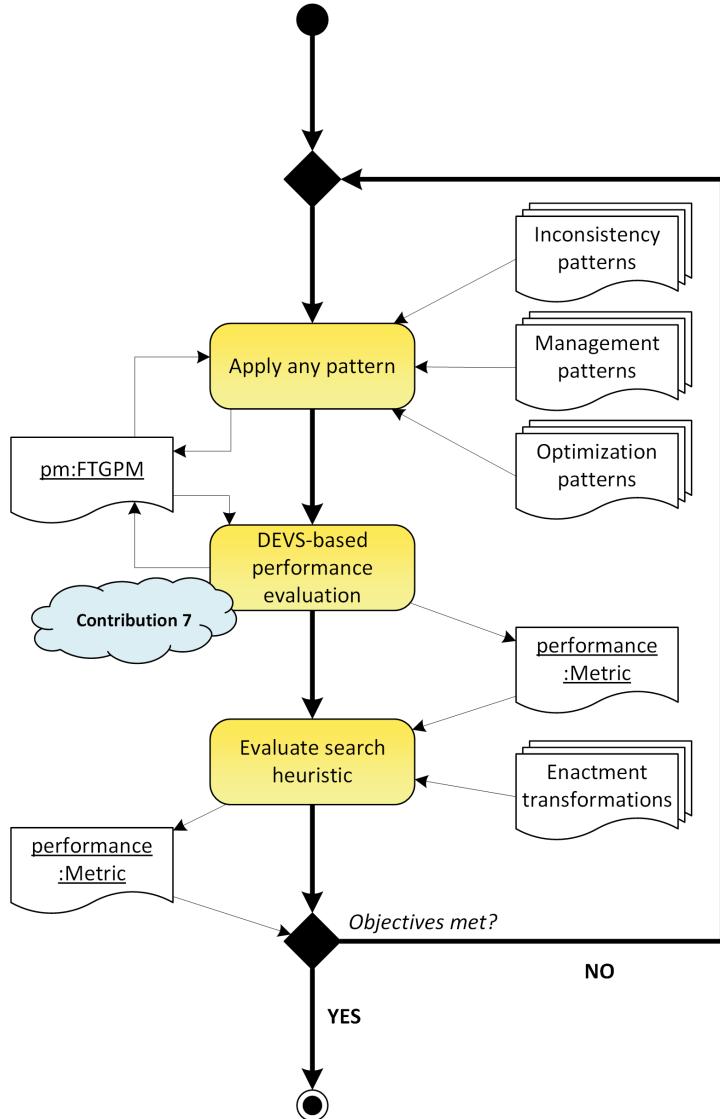


Figure 1.3: Overview of the *Off-line inconsistency management* compound activity, depicted as an FTG+PM.

# Chapter 2

## Background

In this chapter, we give an overview on the foundational concepts related to the broader scope of this PhD thesis. In particular, the topics of model-driven system engineering (MBSE) and engineering processes are discussed.

### 2.1 Model-based system engineering

Engineering is the structured process that leads from the requirements to the product. The American Engineers' Council for Professional Development (ECPD) [63] defines engineering as: *the creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.*

The International Council on Systems Engineering (INCOSE) narrows the scope of engineering to *systems engineering* by the following definition. *Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem: Operations, Cost and Schedule, Performance, Training and Support, Test, Disposal, Manufacturing.* [100]

Model-based systems engineering (MBSE), specifically, is a branch of systems engineering which is characterized by INCOSE as a *formalized application of modeling to support system requirements, design, analysis, verification, and validation activities.* [99] Extensive surveys in [98] and [97] define MBSE in a more general way as *a collection of related processes, methods, and tools.*

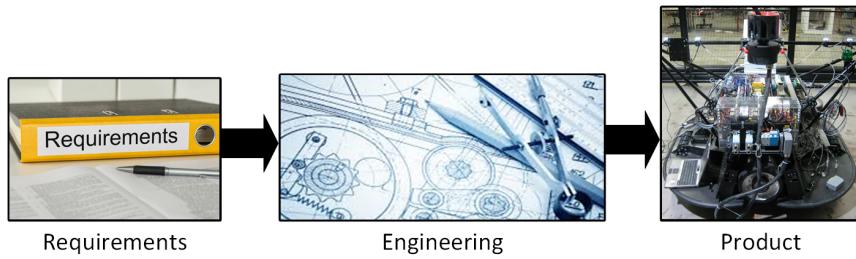


Figure 2.1: General overview of the engineering process.

Figure 2.2 shows a formalized overview of MBSE approaches, depicted by the UML2.0 Activity Diagram notation.

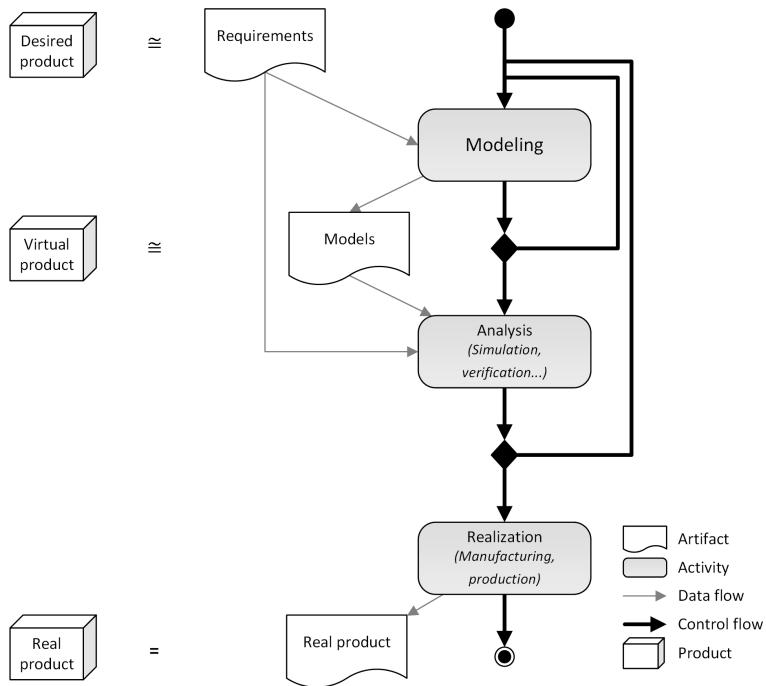


Figure 2.2: Model-based engineering.

In the remainder of Section 2.1, we walk through the concepts shown in Figure 2.2. In Section 2.1.1, the notion of requirements is outlined along with the typical workflow of deriving them from the customer needs. In Section 2.1.2, the notion of the virtual product is discussed. And finally, in Section 2.1.3, we discuss models.

## 2.1.1 Requirements

Requirements, on an abstract level, are the input to the engineering process, and serve as guards for evaluating the correctness of the final product. If the final product meets all the requirements, it is considered a correct product, otherwise it is an incorrect one. A more formal definition is given by the CMMI standard [35], which states, a requirement is:

1. a condition or capability needed by a user to solve a problem or achieve an objective;
2. a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document;
3. a documented representation of a condition or capability as in (1) or (2).

**Example (Requirement).** The system must be able to carry out a mission of at least 8 hours.

Point 3 of the definition emphasizes the need for expressing single requirements in a natural or formal language. Documentation is paramount as the requirements can change, evolve and decomposed into new requirements throughout the engineering process. Agile methodologies, such as Scrum [163] and Kanban [114], in particular, advocate embracing the fact of the changing requirements and instead of trying to reduce the impact of the change, novel methods are needed to deal with the impact of changes appropriately. Lately, this philosophy started to appear in the design of heterogeneous systems as well. [49]

Requirements engineering [105] aims at the proper specification and formalization of the customer needs. Figure 2.3 shows the three typical stages of requirement engineering.

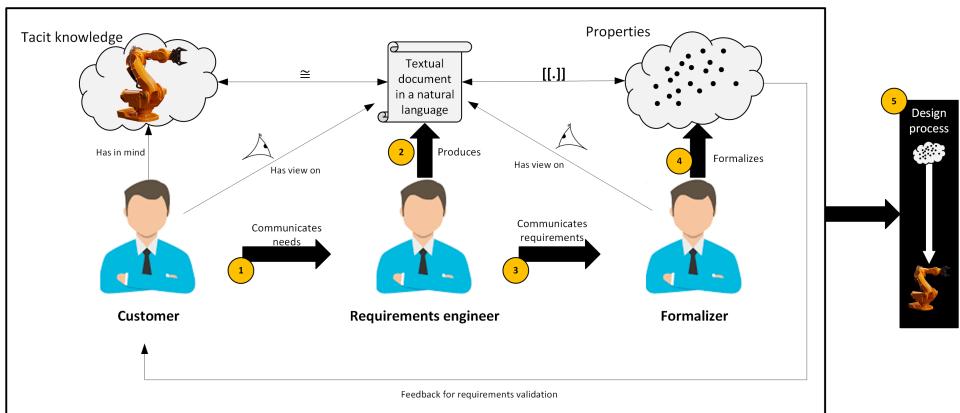


Figure 2.3: The typical workflow of requirements engineering.

Requirements originate from the *tacit* knowledge of the stakeholders. In this phase, requirements only exists in the cognitive mind of the stakeholders, may be communicated

and reasoned about, but they are not documented and therefore, not suitable for driving an engineering process. The task of the requirements engineer is to facilitate the communication with the customer (Step 1) and produce requirement artifacts, typically in a textual form. Requirement documents may follow various templates to make them more formal, e.g. user stories [37] of agile methodologies. This approach, however, is still not formal enough for computer understanding and automated reasoning, but through a structured format, it provides easier understanding for the human stakeholders. Typical approaches include scenario-based [160] and goal-oriented requirements engineering [194].

In a subsequent step, the requirements may be formalized. The task of the formalizer is translating the requirements to properties, by the means of formal languages, formalisms. Pertinent examples of such formalisms include very foundational ones, such as various branches of mathematics (first-order logic, Euclidean geometry, etc), physics (mechanics, dynamics, etc); but also more specific formalisms, such as automatas, Petri-nets, etc. Thanks to the formal nature of these properties, they are suitable for automated (computer-aided) analysis. This also entails detecting inconsistencies between requirements, as discussed in Chapter 4. Conversely, the requirement artifacts used to capture the properties, are in fact essential *models* of the customer needs, and consequently, they become first-class citizens of any MBSE approach. At this stage, the customer may be provided with an appropriate presentation of the derived properties to validate the requirements before the design process commences. [215]

Requirement artifacts are then used in the design process, which aims at satisfying the properties implied by the requirements.

### 2.1.2 Virtual product

Before the actual system, product gets realized, the engineers are concerned with its architecture, parameter and behavior from various aspects. To inspect these, a virtual abstraction of the product is created in order to carry out analyses (e.g. verification, validation, simulation, etc), which would be too costly and too late to be carried out once the final product has been built. The virtual product (VP), therefore, is realized using formal modeling languages, which enable capturing the essence of the final product from *domain-specific* aspects.

There is no universal definition to the notion of the VP, but all the definitions express the intention of *modeling* the end product using computer automation. [93, 25] Atkinson and Draheim [10] refer to the VP as the single underlying model (SUM).

In this thesis, we consider the virtual product as the snapshot of the union of all of the system models *up to* a certain point of the development process<sup>1</sup>, not including (i) the requirements, and not including (ii) the actual, realized end product.

The above convention implies that the VP undergoes significant (and probably frequent) changes throughout its lifeline. These changes are the natural way of engineering a system or product. The motivation behind every change is to converge towards meeting all the requirement previously stated against the system.

### 2.1.3 Models

A model is an abstract representation of another concept, which can be a real thing, or another model.

An early definition of formal models originates from Minsky [130] (1965), who defines a model as follows.

**Model (Minsky).** To an observer B, an object  $A^*$  is a model of an object A to the extent that B can use  $A^*$  to answer questions that interest him about A. It is understood that B's use of a model entails the use of encodings for input and output, both for A and  $A^*$ . If A is the world, questions for A are experiments.  $A^*$  is a good model of A, in B's view, to the extent that  $A^*$ 's answers agree with those of A's, on the whole, with respect to the questions important to B.

The same idea, although in a more software-oriented sense appears in *Liskov's substitution principle* [117] from 1987.

**Liskov's substitution principle.** If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

In his foundational book *Allgemeine Modelltheorie* from 1973, Stachowiak identifies three properties that make a model [173]. (Translation given by [132].)

**Mapping** Models are always models of something, i.e. mappings from, representations of natural or artificial originals, that can be models themselves.

**Reduction** Models in general capture not all attributes of the original represented by them, but rather only those seeming relevant to their model creators and/or model users.

**Pragmatism** Models are not uniquely assigned to their originals per se. They fulfill their replacement function (i) for particular – cognitive and/or acting, model using subjects, (ii) within particular time intervals and (iii) restricted to particular mental or actual operations.

Box, in 1979, while investigating the robustness of mathematical models, states:

All models are wrong but some are useful [31].

This is, indeed, in line with Minsky's criterion of  $A^*$  being a good model; and aligns well with Stachowiak's reduction and pragmatism properties.

It was Bezivin in 2004 who suggested the unification principle of models, stating: *Everything is a model* [21]. This is a statement which every MDE practitioner can easily relate to, albeit, a rather philosophical one. Additionally, the principle stems from the software

world and its grammar-based, essentially closed semantics. Jean-Marie Favre points out the shortcomings of this view by questioning the meaning of the “is-a” relation [67]. Eventually, Favre comes to the conclusion that *nothing is a model*, but rather: a model *plays a role* of the modeled entity.

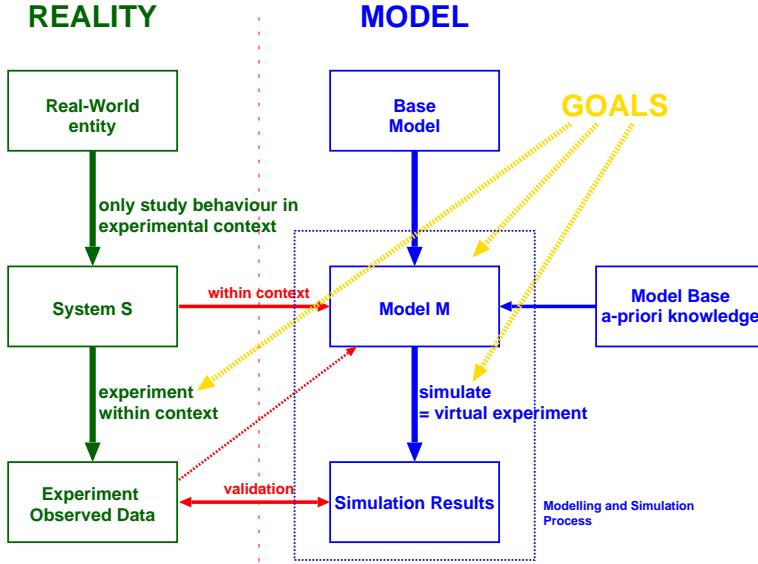


Figure 2.4: Modeling and simulation concepts. [202]

Figure 2.4 presents the concepts of modeling and simulation, as introduced by Vangheluwe et al [202] and Zeigler [216]. In this interpretation, the *Model* is an abstract representation of a *System* within the context of a given *Experimental Frame*. The *Model* reflects certain properties of the system's structure and/or behaviour (within a certain range of accuracy). The *System* is a well defined object in the *Real World* under specific conditions, only considering specific aspects of its behaviour. *Experimental Frames* describe a limited set of circumstances under which a system (real or model) is to be observed or subjected to experimentation. The purpose of building the *Model* is to enable virtual experimentation with *System*, e.g. by simulating or debugging the *Model*. This workflow is shown in Figure 2.5.

To leverage the potential of model-based systems analysis, Vangheluwe suggest embracing the principles of multi-paradigm modeling (MPM) [202].

**Multi-paradigm modeling (Vangheluwe).** Model everything explicitly, using the most appropriate formalism(s) on the most appropriate level(s) of abstraction, while also modeling the process(es).

In this thesis, we fully embrace this directive. In order to investigate inconsistencies in heterogeneous settings, we provide means (theory, formalisms, tools) for modeling every aspect of the engineered system, including the semantic links between disparate domain-specific models, thus enabling a novel approach for inconsistency management.

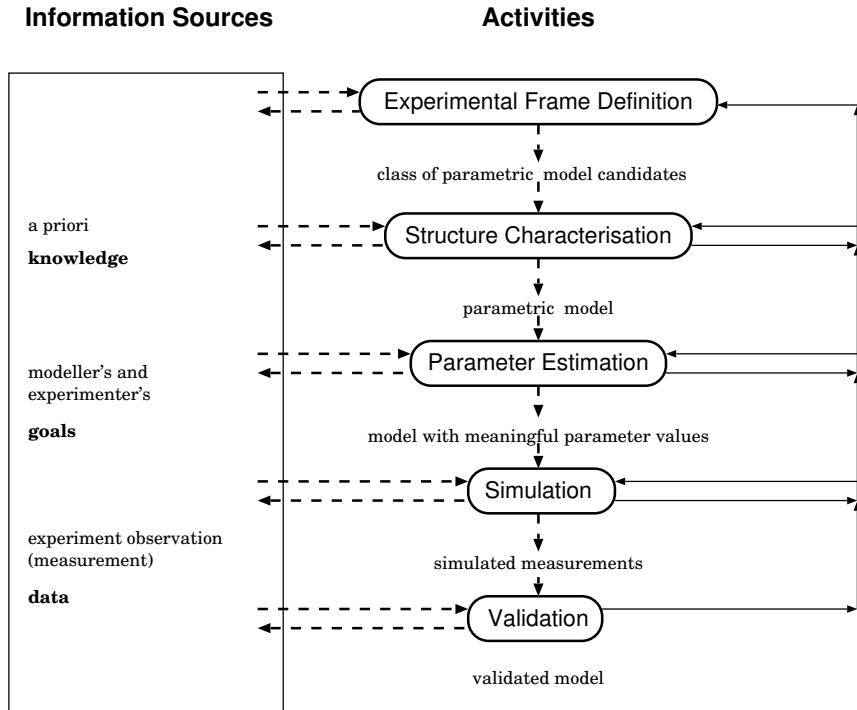


Figure 2.5: Model-based systems analysis. [202]

To support the idea of the “most appropriate formalism” in MPM, *domain-specific modeling languages* (DSL) have been proposed.[74] In contrast with general-purpose modeling languages (such as UML), DSLs aim at relatively small problem domains with restricted syntax and semantics. This results in the model-space of the virtual product being decomposed into very detailed and fine-grained models pertaining to the specific problem.

**Modeling artifacts.** In this thesis, we use the term *modeling artifact* as a general notion of models and views. In accordance with the role-based view of Favre [67], we also allow a modeling artifact to play different roles throughout the engineering process. The two fundamental roles are: *requirement artifact* and *design artifact*. This distinction provides an appropriate framework for the systematic formalization of correctness and consistency between any artifact in the engineering process, as discussed in Chapter 4.

## 2.1.4 Properties and attributes

The term *property* is vastly overloaded already in the computer science domain, and this problem gets exacerbated by the presence of other domains in the engineering process. (Which is exactly the case in the engineering of heterogeneous systems.)

In our terms, a property is a function over a given attribute of the system.

Properties specify the desired values of the various aspects of the system. Properties enforce a set-based reasoning, following open-world semantics. By set-based reasoning, we mean that the *satisfaction* of a property determines whether the system falls into a given set of all the systems or not. By open-world semantics we mean, that upon inferring the satisfaction of a property, the result can be either positive (the property is satisfied), negative (the property is not satisfied), or inconclusive (neither of the previous two can be inferred). If there exists a proof that a property can never get satisfied, the property is unsatisfiable; otherwise it is satisfiable.

**Example (Attribute).** The mass  $m$  of the system, represented as a rational number:  
 $m \in \mathbb{Q}$ .

**Example (Property).** The system is correct if its mass is less than 10 kilograms.  
 $p : m_s < 10$ .

Following the same definitions for a set of properties, *joint* satisfaction and satisfiability can be defined as  $\forall p \in P$  is satisfied.

Evaluating property satisfaction is typically operationalized by carrying out simulations and interpreting the resulting traces; by formal verification; or by testing.

In this thesis, we use properties to characterize the system with. That is, after translating requirements onto the set of required properties, the correctness of the resulting product is always evaluated against the system of properties.

We distinguish between system properties and ontological properties. A system property is any property specific to the given system. An example for such a property could be: *the mass of the system should be less than 10 kilograms* ( $m < 10$  [kg]). Ontological properties, on the other hand, are more generic in the sense, that they apply to every engineered system. An example for such a property could be: *every mass property has a value that is larger than zero* ( $\forall m \in M : m > 0$  [kg]). The conjunction of these two properties results in a well-defined range of accepted values for the mass attribute of the system:  $0 < m < 10$  [kg].

Property-based approaches have been shown to be extremely feasible in detecting inconsistency and incorrectness in the engineering of highly complex heterogeneous systems [203]. In Chapter Chapter 4, we provide definitions of correctness and consistency by the means of properties.

## 2.1.5 Viewpoints, views

The complexity of nowadays engineered systems requires their thorough modeling before the realization of the system. With the growing complexity the types of stakeholders also increases: nowadays heterogeneous systems typically require and interplay between engineers from various disparate domains, such as mechanical, electrical and control

engineering. This diversification of concerns requires breaking down the engineering goals and tasks in accordance with the engineering domains present in the engineering endeavor.

Introducing views and viewpoints to the engineering endeavor supports this approach.

According to the ISO/IEC/IEEE 42010:2011 (formerly: IEEE 1471) standard[101], a *viewpoint* identifies the set of concerns and the representations/modeling techniques, etc. used to describe the architecture to address those concerns. Applying a viewpoint to a particular system results in a *view* of the system.

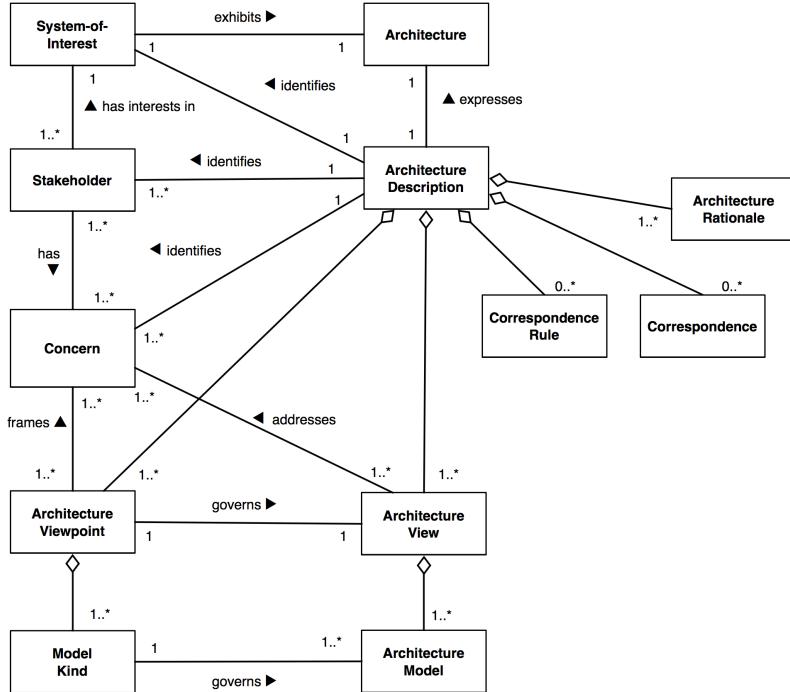


Figure 2.6: The Core of Architecture Description in ISO/IEC/IEEE 42010:2011. [101]

As Figure 2.6 shows, view(point)s are used to investigate the architecture of the system. An *Architecture Description* is a work product used to express the *Architecture* of some *System Of Interest*. The *Concern*'s of the *Stakeholders* will form the eventual *Architecture Viewpoints*, which when applied to an *Architecture Description*, will result in *Architecture models*.

Various architecture description languages exists, and some of them are compatible with the ISO/IEC/IEEE 42010:2011 standard, such as ArchiMate [181] and SysML [139].

Broman et al [32] adapt the general definition of the ISO/IEC/IEEE 42010:2011 viewpoints to the design of cyber-physical systems and propose a framework in which viewpoints are linked to modeling formalisms and to modeling languages and tools. Their framework consists of three elements: viewpoints, which capture the stakeholders' interests and concerns;

concrete languages and tools, among which the stakeholders must make a selection when defining their CPS design environments; and abstract, mathematical formalisms, which are the “semantic glue” linking the two worlds. Corley et al [38] investigate typical scenarios of multi-view modeling, in which they link single/multi – model/view scenarios and identify potential breaches of consistency.

### 2.1.6 Typical scenarios in MVM

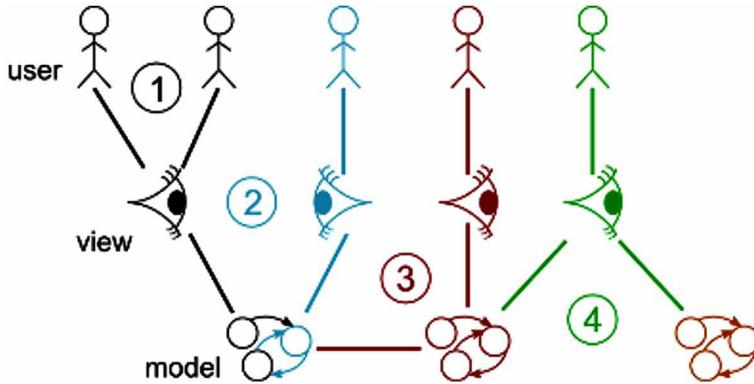


Figure 2.7: Scenarios in multi-view modeling.

Corley et al. [38] identify four typical collaborative modeling scenarios shown in Figure 2.7. In *Multi-User Single-View* settings (Scenario 1), users work on the same view of the model and observe the exact same data in the same concrete syntax. The other three scenarios are characteristically different as they feature multiple views or multiple models, which are the primary cause of emerging inconsistencies. In *Multi-View Single-Model* settings (Scenario 2), users work on different views of the model. The two views may use different concrete syntax. The views are projections of the same single underlying model (SUM) and therefore conform to the same modeling language. Changes in the abstract syntax of one view have to be reflected in the other view in order to retain consistency. The *Multi-View Multi-Model* scenario (Scenario 3) does not assume a SUM, but multiple separated underlying models, which are connected via their semantic domains. Manipulating the models through the respective views causes mismatches in the models themselves and thus, inconsistencies may arise. Finally, the *Single-View Multi-Model* scenario (Scenario 4) is a special form of the Multi-View Multi-Model scenario, in which one view corresponds to multiple underlying models. In practice, collaborative modeling settings typically feature combinations of these scenarios.

The views can belong to different domains, i.e. they may represent various aspects of the SUM in different formalisms and on different levels of abstraction. A typical example is the mechanical design of a complex system where multi-body models and more detailed finite element models are used to reason about different aspects of the same virtual product.

Inconsistencies arise due to the *shared elements* between views: as one view changes an overlapping element, the change has to be propagated to the other views that share the same

element, otherwise an inconsistency will occur [71, 83]. These related elements are not necessarily of syntactic nature, but they may exist in the *semantic domain* of the SUM as the *ontological properties* of the system, e.g., safety, energy-efficiency, autonomy, etc. The conformance of the system to such properties can be evaluated by simulations or model checking.

To reason about characteristics of the semantic domain of models (such as overlaps and linked properties), an explicit representation of the semantic domain has to be provided.

## 2.2 Processes

Process and workflow modeling is an extensively researched area and is typically considered with a specific application domain. Business processes, engineering processes and software processes are among the pertinent examples which rely on a notion of the ordered activities, but define a process (or) workflow differently. Examples include: purchase order, tax declarations, insurance claims process, organizing engineering activities into a product development project.

In its broader sense, processes are structured depictions of a work consisting of multiple steps in order to achieve a desired state, e.g. delivering a product. The Oxford dictionary defines [145] the process as

a series of actions or steps taken in order to achieve a particular end.

More specifically, a process definition specifies (i) which steps are required, and (ii) in what order they should be executed. [188] On a deeper look, a process definition consists of three elements:

**Tasks (step, process element):** A logical unit of work, e.g., typing a letter, stamping a document, checking personal data

**Conditions (state, phase, requirement):** A condition is used to determine the enabling of a task.

**Subprocesses:** Use of previously defined processes, and the ability to reuse frequently occurring processes

**What's in the name?** Processes and process-like structures are referred in many ways, depending on the domain and discipline the process is situated within. In this work, we use the following terms interchangeably: process, workflow, procedure, routing definition.

The Accreditation Board for Engineering and Technology's (ABET) defines [3] the *engineering* process as

a decision making process (often iterative) in which the basic sciences, mathematics, and engineering sciences are applied to convert resources optimally to meet a stated objective. Among the fundamental elements of the design process are the establishment of objectives and criteria, synthesis, analysis, construction, testing and evaluation.

According to Weske [211],

a business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations.

In this research, we investigate processes more of the engineering type. But the above definition is still applicable. Engineering processes consists of engineering activities that are performed in order to realize an engineering goal: the engineered system.

We define processes in a more general manner to fit our needs.

**Process.** A process is a partially ordered set of activities, with the ordering operator  $a_i < a_j$  between two activities. The semantics of the operator can be given by the various before-after semantics of the Allen-algebra [5], usually meaning  $a_i$  is finished before  $a_j$  is started. If not  $<$  relationship is present between two activities, the two activities can be executed in parallel.

This definition is sufficient for our purposes, as it sheds light on the very basic structure of the processes we consider. It defines activities as the foundational elements of a process; and allows two respective properties to either define an ordering or leave that restriction and implicitly allow parallelization.

**Process management (Weske [211]).** Process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of processes.

Processes play an important role in various engineering domains. Pahl et al [146] in 1984 provide a systematic overview of the design processes in traditional engineering. These processes are characterized by rigidity the separation of engineering concerns, and a resulting siloed approach. With the advent of modern day software engineering, the related engineering processes also became a subject of improvement. With the engineering product becoming a purely virtual artifact, engineers became more experimental with their processes. Starting from the waterfall model [149], the Rational Unified Process (RUP) [108] gained wide-spread visibility all around the software domain. Later on, to overcome the inherit rigidity of these process models, the processes have been shifted towards the iterative and incremental nature. [28] Enabled by the mechanisms of MBSE, the V-model and the subsequent Y-model have been popularized in safety-critical systems engineering. [34] Modern day software methodologies often build on the concept of *agile* [11]

project management techniques.

Business processes constitute yet another typical type of processes. They are situated on the higher level of the business hierarchy, typically governing strategic operations. Due to the drive from the business domain and the instantly quantifiable gain they provide, business processes were the prime candidates to benefit from modeling and analysis of performance.

The various fields related to processes and important for this thesis are: process modeling and process enactment.

### 2.2.1 Modeling

The modeling of the process is paramount in modern business and engineering settings. By viewing the process as a resulting artifact of a planning or pre-planning work phase, it becomes obvious, that the same motivations apply as in the case of any software or system being engineered.

There are various process modeling *formalism*, typically tailored for specific domains. Higher level modeling formalisms are typical in business process scenarios. Typical examples include BMPN2.0 [140] and BPEL [138]. These process models are often executable, so that the execution of the process can be automated to minimize the human effort. Medium level formalisms are more general purpose and versatile, and are typically executable. Typical examples include FTG+PM [121], Activity diagrams [52], YAWL [186], MWE2 [58] or Apache Maven [8]. There is a third category of process modeling formalisms: the ones without a clear notion of the process. These formalisms focus on too low levels of abstraction to trivially see them as a process language, yet, they are bone fide process modeling formalisms. Examples include Petri-nets [134], Statecharts [85], scheduling models (e.g. Gantt-charts [212]), data dependency models with resolution semantics.

The notion of the process can be either explicit or implicit. In the former case, at least the control and data elements of the process are made explicit. In the latter case, either the control or the data, or both are only implicit. This is typical when a formalism is more concerned with the state of the work items instead. The traces of the execution, however, correspond to an underlying process model.

The intent of the process can be descriptive (what/how is being done), prescriptive (what/how must be done) or proscriptive (what/how must NOT be done). Typically, a combination of these intents is required to properly model a process.

### 2.2.2 Enactment

Process enactment is commonly defined as the use of software to support the execution of the operational process. [143, 187, 109]. As opposed to process execution, enactment is more broader and considers human interaction. Furthermore, human interaction is typically viewed as dedicated, special activity with its own characteristics and modeling standards [137].



# Chapter 3

## State of the art

In this section, we present **Contribution 1** of this thesis: a systematic overview of the state of the art. The majority of this section has been published in [54].

In this section, we review the state of the art on the two major fields related to this thesis: (in)consistency management and process engineering. Both areas have been thoroughly researched by multiple authors of multiple domains, constituting a vast body of knowledge. We have conducted a systematic literature review on (in)consistency management, keeping our focus on the applicability and scalability of the techniques in heterogeneous system engineering settings. Our findings were described in greater details in our technical report [54], including the steps to the systematic literature review, such as the selection procedure, inclusion/exclusion criteria, etc. In Section 3.1, we summarize these findings. In Section 3.2, we give an overview on the state of the art of process engineering, i.e. process modeling, optimization, enactment, performance analysis and service integration.

### 3.1 (In)consistency management

What are inconsistencies after all? How do we formalize them? How do we deal with them? In the past years multiple contributions have been made to answer these questions. In this section, we focus on the following questions.

- **What methods, techniques and tools are available for the management of consistency in a mechatronic and CPS design process?**

Many of the contributions to the field have been made in the context of pure software systems. As opposed to this context, our main focus lies on the design of mechatronic and cyber-physical systems in particular.

- **What are limitations of the state-of-the-art?**

Identifying limitations and rather, the shortcomings of the state-of-the-art is a vital part of this report. We rely on this information to fill the gaps in what we believe to be an efficient approach for handling inconsistencies.

- What makes consistency management in a mechatronic and CPS context different from the software engineering industry in general?

### 3.1.1 Selection method

Our selection method was based on the guidelines described in [103].

As shown in Figure 3.1, the process started with selecting the appropriate search terms (Section 3.1.1.1) and sources (Section 3.1.1.2). The Cartesian product of these inputs parametrized the search process. To successfully qualify to be a candidate, every paper went through the exclusion criteria (Section 3.1.1.3).

Selecting the appropriate papers took two rounds of reviewing (Section 3.1.1.4). Afterwards, the Google Scholar results were included by making sure not to introduce any duplicates.

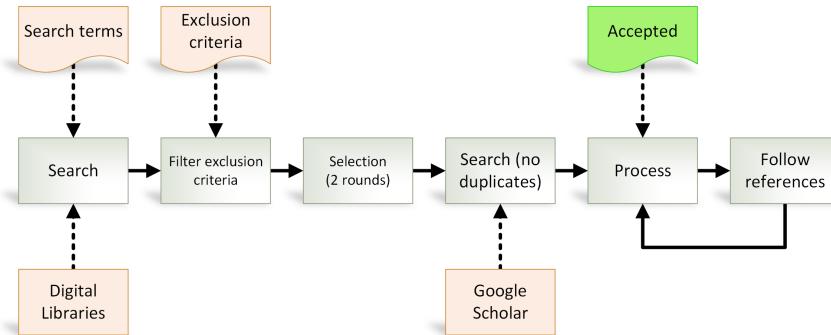


Figure 3.1: High-level workflow of searching, selecting and processing.

Finally, we processed the selected papers and in parallel, followed the interesting references.

#### 3.1.1.1 Search terms

To constraint scope of search for papers, the following search terms have been selected:

- consistency management, inconsistency management;
- overlap detection;
- inconsistency detection;
- inconsistency resolution.

Additionally, the following extra search terms have been also used:

- multi-paradigm model(l)ing, multi-view model(l)ing;
- mechatronic design;

- concurrent engineering;
- tool integration.

### 3.1.1.2 Sources

We used the search terms to look for relevant publications from the following sources:

- digital libraries: ACM, IEEE, Springer;
- Google Scholar;
- manual search based on top cited articles (authors and cited by).

(We plan to extend our current work by also including domain-specific sources, such as journals of software engineering, mechanical design, etc.)

First, the digital libraries have been searched through. The top 15 papers for every search term have been accepted as candidates for the further evaluation rounds. After selecting the top 15 publications, the year of publication has been constrained to the past two years as newer publications tend to sink lower in lists ordered by relevance. That makes a total set of candidates of 20 per source and per search term. The total amount of papers included for the first round was approximately 200. (With duplicates.)

Additionally, we searched through Google Scholar after the review and selection step.

### 3.1.1.3 Exclusion criteria

We defined a set of exclusion criteria against the candidates: non-English and short papers got rejected immediately.

### 3.1.1.4 Review and selection

The detailed review process is shown in Figure 3.2. To review the approximately 200 papers, papers had been divided and assigned to one of the two reviewers. In the *1st round* of evaluation, three groups were formed.

**Accept** A paper found to be relevant based on its abstract, got immediately accepted for further processing.

**Reject** A paper found not to be relevant based on its abstract, got immediately rejected.

**Maybe** If the relevance of the paper could not be decided, the paper was assigned to this group and processed in the *2nd round* of evaluation.

To evaluate which group a paper belongs to, we investigated how many of the following properties a paper has.

- It gives an overview.
  - It's a survey / systematic review. (Strong accept)
  - Presents a classification of the types of (in)consistency problems...

- in the domain of mechatronics or maybe avionics/automotive. (Immediate accept)
- Presents a technique...
  - for overlap detection.
  - for inconsistency detection.
  - for resolution.
  - the technique is supported via a (prototype) tooling.

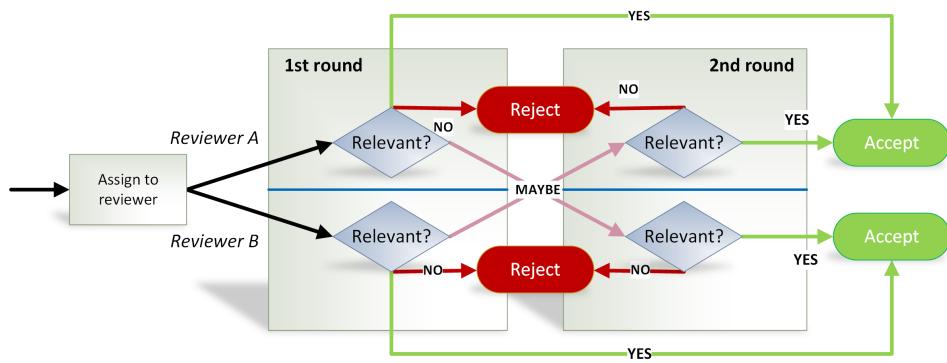


Figure 3.2: Detailed selection process.

In the *2nd round* of evaluation the papers reviewed and assigned to the “Maybe” group, were evaluated by the other reviewer. This time, the evaluation was carried out based on not just the abstracts, but the introductions as well. Proving to be relevant, the paper got eventually accepted, otherwise rejected.

### 3.1.1.5 Results

At the of the selection process, approximately 70 papers have been identified to be interesting to be actually incorporated in our survey. This number has been extended by followed references during the processing phase.

### 3.1.1.6 Threats to validity

During the search and selection phase, we identified the following threats to the validity of this survey.

**The overloaded meaning of “inconsistency”** Inconsistencies are natural phenomena in several domains, not just in multi-model settings. Typical examples include: inconsistencies violating the ACID principle in relational database management systems, and well-formedness violations in software design. This overloaded nature of the term makes it hard to filter the works relevant to our scope.

**Semantic gaps between different domains** Even in closely related domains, the definition of inconsistencies is not clear. As a typical example, in pure software modeling, and especially in the case of UML modeling, inconsistencies are defined solely as well-formedness violations, i.e. a discrepancy arising from the linguistic properties of the domain. In a true MPM setting, although, semantic inconsistencies are also considered as discrepancies arising from the overlaps between various formalisms, domains.

**Hidden domains** Although our survey focused on multi-model setting in general, this proposition also implicitly assumes the involved domain actually employs explicit modeling techniques. If there is no link made between a domain and explicit modeling, a domain may remain hidden. As a typical example, we reached as far as the rather exotic domains of “concurrent engineering”, which is not linked directly to MPM, but we had a priori knowledge about potential links. We plan to gradually extend our technical report as hidden domains get discovered.

### 3.1.2 What are inconsistencies?

There is no unified definition of “inconsistency”, as it is always specific to the characteristics of the given domain, setting and task. According to the Merriam-Webster dictionary [1]:

**Inconsistent.** Not always acting or behaving in the same way; having parts that disagree with each other: not in agreement with something.

Spanoudakis et al [172] define inconsistencies specific to software models as

a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable.

Herzig et al [90] also add that

an inconsistency is present if two or more statements are made that are not jointly satisfiable,

with the typical examples of

failure of an equivalence test, non-conformance to a standard or constraint and the violation of physical or mathematical principles.

### 3.1.3 The origin of inconsistencies

Investigating their origin, gives a better understanding on the nature of inconsistencies. Some authors claim the **imprecise or vague semantics** of modeling languages being a potential source of inconsistencies:

Another source of inconsistency is the imprecise semantics of the UML. [95]

More frequently, however, inconsistencies arise in settings featuring **multiple views** on the same virtual product and the mismanagement of the information between these.

(...) inconsistencies arise because the models overlap – that is they incorporate elements which refer to common aspects of the system under development – and make assertions about these aspects which are not jointly satisfiable as they stand, or under certain conditions. [172]

Views conforming to these viewpoints are highly interrelated due to the concerns addressed overlapping. These interrelations and overlaps can lead to inconsistencies. [90]

(...) if different views of a system have some degree of overlap, how can we guarantee that they are consistent, i.e., that they do not contradict each other? [156]

Additionally, the **evolution** of single model artifacts can also lead to inconsistencies. Hennenberger et al [88] note that

large design models may contain thousands of model elements. Designers easily get overwhelmed maintaining the correctness of such design models over time. Not only is it hard to detect new errors when the model changes but it is also hard to keep track of known errors. In the software engineering community this problem is known as a consistency problem and errors in models are known as inconsistencies.

### 3.1.4 Managing inconsistencies

Relational database management systems and version control systems overcome the problems of inconsistencies by *avoiding* them in the first place. In multi-view settings, however, this cannot be always guaranteed and the *allow-and-recover* style of management is more appropriate. Once inconsistencies are encountered, their root causes are required to be analyzed in order to manage them properly. Finkelstein et al [70] notes that

Rather than thinking about removing inconsistency we need to think about "managing consistency".

In our view, however, it is more appropriate to reason about managing **inconsistencies**, since we assume that inconsistencies cannot be avoided and the real challenge is to deal with them *after* they are encountered.

### 3.1.5 Features and patterns of inconsistency management techniques

In this section we define a structured taxonomy for discussing various inconsistency management (ICM) techniques and approaches. In our taxonomy, we combine those of Spanoudakis et al [172] and Hanmer et al [84] and characterize inconsistency management techniques and by their *features* and the *patterns* used for of implementing those features. (For example, using *Pivot models* (Section 3.1.7.1) is a typical pattern of the *Representation* feature (Section 3.1.7.1).) While the usage of features assumes a closed-world, and thus gives a rigid structure to our view on inconsistency management techniques in general, we assume an open-world in case of patterns of implementing those features, since it is not feasible to exhaustively enumerate and classify all of the patterns.

Not every inconsistency management technique addresses the same challenges in inconsistency management. While some of them focus on specific sub-problems (such as detecting inconsistencies), others approach inconsistency management from a holistic view and attempt to come up with a framework for inconsistency management. Based on our survey of the state of the art, we compiled a feature model for inconsistency management techniques, shown in Figure 3.3.

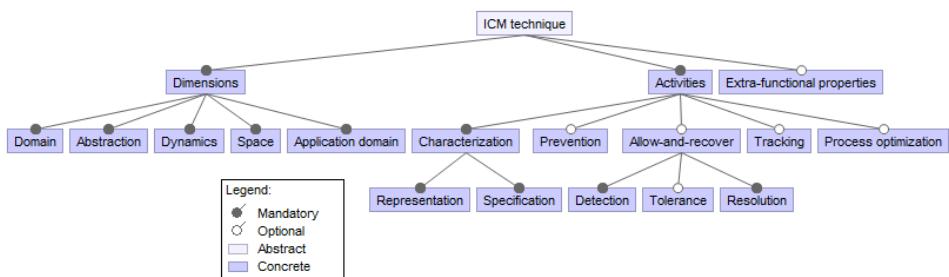


Figure 3.3: Feature model of the inconsistency management techniques.

The two required features of inconsistency management techniques are their *dimensions* and *activities*. Dimensions are used to investigate *what* an inconsistency management technique attempts to tackle, while activities tell *how* they achieve that. Additionally, the extra-functional properties of an inconsistency management techniques may be also interesting in some cases, as they tell *how good* a technique with respect to a specific metric (such as performance, precision, etc).

Feature	Meaning
Dimension	What?
Activity	How?
Extra-functional property	How good?

Table 3.1: Features of inconsistency management and their interpretation.

### 3.1.5.1 Dimensions

Dimensions identify *what* a given inconsistency management technique attempts to cover. More specifically, what types of inconsistencies it considers. Table 3.2 summarizes the dimensions considered at this place.

Dimension	Patterns
Domain	Linguistic Semantic
Abstraction	Horizontal Vertical
Dynamics	Static Evolution
Space	Intra-model Inter-model

Table 3.2: Inconsistency management dimensions and their patterns.

#### Space

Depending on whether the considered inconsistencies are restricted to just one single model or are allowed to extend over multiple models, intra-model and inter-model inconsistencies are distinguished. [120] Intra-model inconsistencies occur in one single model, while inter-model inconsistencies can be observed between different models.

Due to the prevalence of multi-view settings in the mechatronic domain, techniques feasible for inter-model (inter-view) inconsistency management are the preferred ones. Since inconsistency management techniques suitable for inter-model situations are also suitable for intra-domain cases, we only consider approaches capable of inter-model inconsistency management and do not investigate this dimension further.

#### Domain

The domain defines whether the considered inconsistencies are situated in the *linguistic* or the *semantic* domain of a model. Linguistic inconsistencies arise from syntactic discrepancies and are equivalent to well-formedness violations, as known in software modeling.

Semantic inconsistencies, on the other hand, are results of overlaps between different domains and formalisms involved in the modeling process.

### Abstraction

Depending on whether an inconsistency can extend over various levels of abstraction, Lucas et al [120] distinguish between *horizontal* and *vertical* inconsistencies. While horizontal inconsistencies occur between models on the same level of abstraction, vertical ones are associated with models on different levels of abstraction.

### Dynamics

An inconsistency management technique can static vs evolution] Depending on whether the notion of inconsistencies supports reasoning between different versions of the same model, inconsistencies may be defined either as static or evolutionary.

### Discipline

Also known as the *application domain* of a technique, the discipline identifies the scope an inconsistency management technique can be successfully applied within. The majority of the state of the art considers only software engineering as an application domain.

#### 3.1.5.2 Activities

Activities define *how* an inconsistency management process is structured. We mainly follow the classification of Spanoudakis et al [172], but further extend it as presented later. Our classification of inconsistency management processes is shown in Figure 3.4.

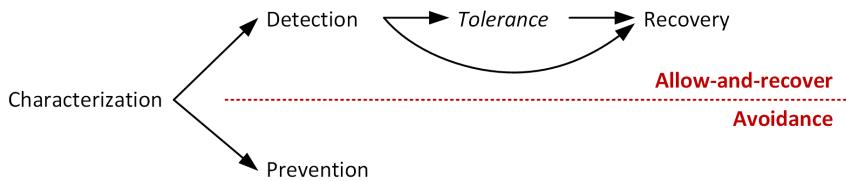


Figure 3.4: Conceptual overview of inconsistency management styles and activities.

### Characterization

Characterizing inconsistencies is the necessary first step to every inconsistency management process. Its role is twofold. First, the inconsistency rules defined in this step serve as inputs for the subsequent activities (detection or avoidance). Second, the employed patterns of inconsistency characterization, heavily influence the choice on the patterns of the subsequent steps.

Different authors approach this process with various granularity and see the role of characterization (often referred as *inconsistency definition*) different. Spanoudakis et al [172] do not emphasize the role of proper inconsistency characterization, while Van Der Straeten [190]

acknowledges characterization as the foundational first step towards a managed approach towards handling inconsistencies.

	<b>Characterization</b>	<b>Detection</b>	<b>Resolution</b>	<b>Prevention</b>
Adourian [4]	●	●	●	○
Balzer [12]	●	●	○	○
Becker [16]	●	○	○	○
Bhave [24]	●	●	○	○
Bhave 2 [23]	●	○	○	○
Blanc [27]	●	●	○	○
Dahman [40]	●	●	●	○
Easterbrook [55]	●	●	○	○
Egyed [60]	●	●	○	○
Engels [62]	●	●	○	○
Gausemeier [78]	●	●	●	○
Giese [80]	●	●	●	○
Giese 2 [79]	○	○	○	●
Hamlaoui [61]	●	●	○	○
Hehenberger [88]	●	●	○	○
Herzig [89]	●	○	○	○
Hessellund [92]	●	○	○	●
Le Noire [113]	●	●	○	○
Lopez-Herrejon [118]	○	●	○	○
Lopez-Herrejon 2 [119]	●	●	○	○
Mens [128]	●	●	●	○
Nentwich [136]	○	○	●	○
Oka [141]	●	○	○	●
Qamar [150]	●	●	○	○
Quinton [152]	○	●	○	○
Shah [167]	●	○	○	○
Stolz [175]	●	●	○	○
Van Der Straeten [192]	●	●	●	○
Van Der Straeten 2 [191]	●	●	●	○

Table 3.3: Overview on the state of the art. (Legend: ○ – no support; ● – possible support, not emphasized; ● – dedicated support)

### **Detection**

In non-prevention style techniques, inconsistent situations have to be detected based on the characterization of inconsistencies in order to execute the proper resolution actions. The patterns of characterization heavily influence the patterns of detection.

### Resolution

Once an inconsistent state is detected, it is usually expected to be resolved. Although Finkelstein et al [70] notes, that managing consistency means not only removing the detected inconsistencies, but also

(...) preserving inconsistency where it is desirable to do so.

### 3.1.6 Summary of the state-of-the-art

Table 3.3 briefly summarizes the features supported by the state of the art techniques considered in our survey. The patterns of these techniques are classified and presented in the subsequent sections.

### 3.1.7 Characterization patterns

In terms of characterization, we distinguish between two important aspects:

- representation, i.e. the form (in)consistency rules are defined in, and
- specification, i.e. the way (in)consistency rules are defined.

Typical representation patterns include using *naming conventions*, *graph-like formalisms* and *pivot model based* techniques. Other representation techniques are also discussed in the following section. Concerning specification, most of the available techniques are based on and involve *human inspection*, therefore we only mention those employing a semi-automated or automated technique.

#### 3.1.7.1 Representational patterns

##### Naming conventions

According to Spanoudakis [172]:

the simplest and most common representation convention is to assume the existence of a total overlap between model elements with identical names and no overlap between any other pair of elements.

Hessellund et al [92] investigate inconsistencies in a setting where multiple DSLs are involved in the design process. The technique aims to represent, check, maintain constraints, and that in an avoidance-like fashion. An EMF extension framework, SmartEMF is used which employs Prolog to achieve inconsistency management. Referential integrity is mentioned as a typical (general) type of inconsistencies.

Oka et al [141] identify different types of relationships between (i) two components, (ii)

composite and component objects, (iii) two composite objects. The technique has three main elements:

- update operation patterns - there are three of them: (a) not allowed update, (b) allowed update but no change propagation, (c) allow-and-propagate,
- modification rule matrix - specifies how the previous ones are applied: for every relationship type a pattern is chosen,
- modification algorithm - orchestrates the interplay between the matrix and the rules

The authors do not assess the performance of the framework and there is no known follow-up project or tool. The applicability of the technique seems to be very constrained.

### **Dependency graphs**

The second group of characterization techniques by Spanoudakis [172] are the *shared ontologies*. This approach requires the authors of the models to tag the elements in them with items in a shared ontology. The tag of a model element is taken to denote its interpretation in the domain described by the ontology and, therefore, it is used to identify overlaps between elements of different models.

We found that this technique is hardly ever encountered in mechatronic settings. Instead, graph-like formalisms are used to depict overlaps between various models.

A number of techniques employs a specific subset of dependency models in form of TGGs. [162]

Becker et al [16] describe requirements and the characteristics of tools for inter-model consistency management. Most notably, they identify functionality (m–n relations between objects the tools needs to install), operation mode (incremental preferred over batch), direction (bidirectionalality is preferred) and adaptability (support for different domains) among others and identify TGGs as a good fit with these requirements.

Adourian et al [4] use an explicit correspondence model to relate elements of different models to each other. Unidirectional change propagation can support *interleaved* evolution, where no conflicts are introduced in the separate views of the system. Bidirectional change propagation can support *parallel* evolution, where parallel and conflicting changes are allowed to be introduced. Triple-graph grammars (TGG) are used as a theoretical underpinning to the technique. It is thanks to the (computationally) non-causal nature of Modelica as well as its modularity that an almost one-to-one correspondence between geometry models and dynamics models can be found. Without non-causal models, we would have to associate many causal models with a single geometric model.

Gausemeier et al [78] propose using a cross-domain system model for consistency management, based on TGG. This view of the system is used in conceptual design. The conceptual view (active view), contains system elements and shows the information and energy flow in the system. After conceptual design, all views of the system (in siloed design for example) keep relations with this domain-spanning model. Model transformations are used

to generate the domain-specific views from the conceptual model and to keep the models consistent.

Giese et al [80] focus on bidirectional model synchronization as an important technique in MDSD. They provide an algorithm based on the triple graph grammar formalism (TGG). While their previous work [81] optimizes for a single change, there might be compound changes to be synchronized. The work coins the problem of multiple transformation steps and the need for a formal framework for reasoning about algebraic-temporal structures of these.

#### *Mechatronic/CPS settings.*

Qamar et al [150] present a technique specifically for the mechatronics domain, which considers structural and parameter type dependencies between different domain-models in order to provide consistency between different views. This provides an ability to traverse between different views of the system, as well as maintaining consistency between those views. Inter-domain relationships are established via SysML as a pivot model, between a mechanical model (Solid Edge) and the dynamic analysis model (SimScape/SimMechanics).

Other authors focus on the operational side of dependency modeling.

Mens et al [128] expresses inconsistency detection and resolution as *graph transformation rules*. The dependency analysis of these enables appropriate ordering and refactoring of inconsistency rules. The analysis is carried out using the critical pair analysis algorithm. This analysis can be exploited to improve the inconsistency resolution process, for example, by facilitating the choice between mutually incompatible resolution strategies, by detecting possible cycles in the resolution process, by proposing a preferred order in which to apply certain resolution rules, etc. The approach focuses on structural inconsistencies and therefore, it falls short to tackle the problem of semantic inconsistencies.

Egyed [60] presents an *incremental technique* to detect and keep track of inconsistencies and a prototype tooling to support the approach. The author claims that even “very large industrial” models can be maintained efficiently. The technique identifies the consistency rule instances to be checked upon a model change, based on a *scopes of the changes*. The rules are defined manually and cover structural constraints only; this technique, therefore does not support reasoning over semantic inconsistencies.

Dahman et al [40] investigate the problem of consistency management in the domain of business processes. They propose incremental model transformations to mend evolutionary consistency issues. The case study is the consistency between a BPMN model and a Component-Based Model. Update mechanisms on one model are primitive updates: addNode, insertEdge, dropNode, deleteEdge, setLabel, setSource, setTarget, setIndex. A synchronization algorithm specifically for the BPMN to SCA model is specified in the

paper. The characterization is achieved on meta-model level, and could be used for keeping analysis models up-to-date.

Apart from inconsistencies of static dynamics discussed above, evolutionary inconsistencies can be also supported by dependency graphs.

Hamlaoui et al [61] recognize the infeasibility of depicting virtual product by one huge model and propose a network of related models, which provides a global view of the system through a correspondence model. The approach is claimed to be domain independent. Two types of model evolution are presented: adaptation (between model and meta-model) and co-evolution (on the same level of abstraction). Changes (add, modify, delete) in one model trigger changes in other model(s). An Xtext-based textual DSL is used for defining the correspondence model. Domain specific and -independent structural changes can be detected; semantic changes can be handled via the suggestions provided to the human user, although the paper does not provide examples on this scenario.

### **Pivot models**

A special case of the graph-based representation is the usage of pivot models, which is more frequently used in design of mechatronic and cyber-physical systems. Pivot models act as an intermediate language to transform models into each other.

Shah et al [167] present SysML as a pivot model for system engineering and the related concerns of tool integration. SysML is used as the pivot where also the other languages (abstract syntax) are included by using profiles. Model transformations are used to generate the domain-specific models in the different tools form the SysML model or vice versa. SysML parametric models are used to model the dependency relations within the pivot model. Because of structure changes (topological changes), this would have to be remade every time the structure changes. For this, multi-aspect component models are used. These instantiate a correct component and parametric relations when the topology changes. This information is domain-specific. The paper also provides a case study where the approach is applied to a log splitting machine.

Bhave et al [23] describe an architectural approach to reasoning about relations between heterogeneous system models. A pivot model (the “run-time base architecture”) is used to associate related models. Models are related to the pivot model through architectural views, which capture structural and semantic correspondences between model elements and system entities. The component-and-connector (C&C) perspective is used to define the architecture. The C&C perspective is extended to efficiently address not just software and computational infrastructures, but also the physical parts of a CPS - this is the base architecture (BA). The BA of a CPS is an instance of the CPS architecture style, which contains all the cyber and physical components and connectors that constitute the complete system at runtime. Architectural views for a modeling formalism are defined as a tuple of (i) the C&C configuration of the view (with types, semantics and constraints); (ii) associations of model elements with elements of CV; and (iii) associations of elements in CV with elements in BA, respectively. The technique is demonstrated through a multi-domain case

study on engineering a quadrotor system. The only type of inconsistencies considered is the one arising when model elements are mapped to the BA in a many-to-many style, which is clearly a structural one. The authors claim that current C&C architectural styles, which focus primarily on software and computational infrastructures, are not comprehensive enough to describe a complete CPS. Although the aim of this paper is not inconsistency management, as this question is addressed in Bhave et al [24] introduce the notion of structural consistency management for CPS. Views of a single system are kept consistent using an architectural base model (the pivot model). This base architecture relates the different views of the system to each other. Well defined mappings between a view and a base architecture manage the different consistencies. Defines “weak consistency” as (a) every component in the view is accounted for in the architectural base model, (b) every communication pathway and physical connection in view should be allowed in the architectural base model. Defines “strong consistency” as weak consistency, plus: every element in the architectural base model must be represented in the view.

### **Operation-based representation**

Blanc et al [27] propose to represent models by sequences of elementary construction operations, rather than by the set of model elements they contain. The key idea of the approach, to uniformly detect structural and methodological inconsistencies is, that it relies on elementary model construction operations instead of the model elements themselves. Change operations are motivated by MOF: create, delete, setProperty, setReference. Structural and methodological consistency rules can then be expressed uniformly as logical constraints on such sequences. Structural and “methodological” inconsistencies can be detected. Methodological rules constrain the overall construction process and the authors claim this being the core contribution of the paper. The approach supports multi-model and multi-level modeling. For structural constraints, they define operation pairs, where the second operation cancels out first one. For example, creating a model element and subsequently deleting it.

Le Noire et al [113] propose an approach that represents models as a sequence of operations. The Praxis tool is used as a (meta-model independent) tool for operation based model management, with PraxisRules for consistency constraint definition. The textual DSL expresses consistency constraints in terms of operation primitives and also allows complex change definitions. The search for inconsistencies is performed on the subset of the model that was modified since the same inconsistency check was run last time. The provided use case builds on the ARCADIA framework of Thales. The approach allows modeling both structural and semantic inconsistencies.

Stolz et al [175] present a notion of potentially re-orderable model transformations to track the semantic dependencies of the different modeling steps. The technique assumes model evolution on various meta-levels and that specific model elements may contain different data on different levels. Both structural and semantic issues are considered when change propagation fails. The method starts from an empty model which evolves via primitive operations: add, update, delete. Prerequisites are assumed to be captured in first order logic (e.g. OCL in UML settings), which imply a set of model elements as dependencies. These can be mapped to other parts of the model in the time and scope of a transformation. “Proof

obligations” are used to keep models in a semantically consistent state: transformations generate new proof obligations and the user must prove these. These are stored as the part of the dependency model.

### **Ontologies**

Hehenberger et al [88] present an approach for consistency checking of mechatronic design models by using domain ontologies. Ontologies are meant to be a “structured and organized” way to represent domain knowledge and therefore, enable reasoning over multiple domains. The approach explicitly aims to managing semantic inconsistencies.

### **Rule based representation**

Van Der Straeten et al [191] argue that inconsistency resolution is the key enabler of refactoring as a software engineering technique. Inconsistencies emerge as intermediate states are traversed during a complex refactoring step; the final state is reached via gradually eliminating these inconsistencies. The idea is highlighted by the step-by-step execution of the Move Operation refactoring activity. The paper also presents a rule-based technique for rule-based inconsistency resolution. The authors observe that the same inconsistencies occur in multiple refactoring scenarios and therefore, reusability of resolution strategies is acknowledged as a key factor. Resolution strategies are user-guided. The rule-based approach translates to LHS-RHS structures where the user-specified patterns (LHS) trigger the user-defined resolution strategy (RHS). As the motivation of the paper comes from model/code refactoring, the takeaways are really syntax-oriented; semantic inconsistencies are not addressed. “Object-oriented modeling languages” (and UML, in particular) are the application domain of the presented resolution technique.

Quinton et al [152] supports the evolution of cardinality-based feature models. Specifically, range inconsistencies are handled, i.e. the cases when no product exists for some values of a range of a feature cardinality. The authors identify key scenarios upon add/remove/update/move operations which can lead to range inconsistencies.

### **Formal methods**

Engels et al [62] present a prototype framework for modeling consistency constraints, specifically in UML-based software development. The idea behind the approach is to capture consistency rules by arbitrary formal techniques and then evaluate those over input models. The user has to choose the appropriate formal technique and provide a mapping from and to the input models. As a case study is presented where CSP serves as the formal underpinning. Mappings are captured via graph transformation rules. An extensible catalogue of various consistency problems is provided by the framework to support reusability of (in)consistency concepts.

Giese et al [79] present an inconsistency *avoidance* technique in the context of mechatronic system design by safe composition of systems.

### **Logic-based approaches**

Van de Straeten et al [192] use description logic to characterize and evaluate inconsistent model states. The approach mainly targets inconsistencies arising during model evolution.

Introduces horizontal and evolution consistency; horizontal defined as: consistency between different models within the same version; evolution consistency defined as: consistency between different versions of the same model. A classification of inconsistencies is provided. Description logic is used to depict ontological relationships among model elements. Loom is used as a description logic tool. Inconsistency management is achieved by a UML profile. Unfortunately, the technique has a strong focus on UML and its applicability in a broader modeling domain is questionable.

Lopez-Herrejon et al [119] pose the research question “where should the fixes be placed?” rather than when and how to fix. Possible configurations are mapped to formulas of propositional logic. Consistency rules are captured as well-formedness rules using logical formulas. Inconsistencies arise due to changes in the feature models, although removing elements is not permitted. A metric called “pair-wise commonality” is used to express how many configurations a feature appears in. Given a faulty feature F, the algorithm selects the features with the highest pwc value, since these are most likely to suffer from inconsistencies in this case. The efficiency of the heuristic is demonstrated via measurements. The repair actions, although, are constrained to adding missing features.

### 3.1.7.2 Specification patterns

The most common approach to specify inconsistency rules is to manually inspect models and make links between the appropriate model elements. Spanoudakis [172] refers to this approach as *human inspection*. The vast majority of the state-of-the-art considered in this report relies on human inspection.

As an attempt for automation, Herzig et al [89] presents a characterization technique based on pattern matching and **similarity metrics** defined between different (graph-like) models. Model characteristics, such as entity and property names, property units, relationship names and cardinalities, etc are used to calculate similarity between sets of different models. This enables identifying overlapping elements which are not necessarily connected via syntactic link, but still represent the same or related information. It builds on the premise that models of similar meaning are represented similarly in terms of naming, structure, dimensions, etc. The characterization rules are derived automatically, although the intervention of a human domain expert is inevitable in this case also. This technique, however, is the most advanced automation technique in terms of specification in the state of the art.

### 3.1.8 Detection patterns

In non-prevention style techniques, inconsistent situations have to be detected based on the characterization of inconsistencies in order to execute the proper resolution actions. Typical patterns are the *human-centered collaborative exploration* and the *specialized forms of automated analysis* [172].

In general, the patterns of characterization heavily influence the operational patterns of detection, i.e. the means the detection is executed by.

### **Graph reasoning**

Techniques employing the dependency graph representational pattern (Section 3.1.7.1), typically rely on the related techniques of graph reasoning, such as graph querying, transformations, pattern matching, etc. Hamlaoui et al [61] use EMFCompare [57] to enrich the correspondence model by change deltas among various models. Giese et al [80] employ the FUJABA framework to transform TGGs in consistency-preserving bidirectional synchronization scenarios. The majority of techniques relying on a graph-based representation pattern also uses some form of model transformations, such as Qamar et al [150] and Adourian et al [4] for change propagation, and Mens et al [128] for model-dependency analysis.

### **Rule-based detection**

Rule-based detection is used in combination with dependency graphs by Becker et al [16]. While consistency rules are defined via TGGs among the related models, the detection achieved by the PROGRESS rule engine.

### **Solver-based detection**

Numerous techniques rely on the solver-based detection pattern, typically using a variant of Prolog for tooling. [92] employ this pattern in combination with naming conventions (Section 3.1.7.1), using SmartEMF, an EMF extension framework, which is actually a Prolog fact base. This fact base captures models in EMF (MOF) terms. Basic EMF operations are extended by propagating changes into the fact base. Constraints are captured by Prolog terms. The evaluation is implemented via higher-order queries (using call predicate). Both Blanc et al [27] and Le Noire et al [113] use Prolog in combination with operation-based representation (Section 3.1.7.1). In the former case, edit operations are recorded and traces are fed to a Prolog engine, resulting in an on-demand, batch-like inconsistency check approach. The latter approach builds on SWI-Prolog. The authors compare the time- and space complexity of the detection phase in operation-based and standard representation patterns: no difference in batch mode has been identified, while in incremental mode “some” advantages have been found. (E.g. batch rules can be used without rewriting them). Quinton et al [152] use SAT solvers to evaluate sequences of edit operations in operation-based representation scenarios. Model changes are translated to the BR4CP/ Aralia language [155] and are fed to a SAT solver.

### **Ontological reasoning**

While there is not much work on it, the ontological reasoning detection pattern fits well with ontology-based inconsistency characterization. Although this pattern is not found in the state of the art, techniques such as Hohenberger et al [88] could employ it.

### **Incremental evaluation**

The incremental evaluation pattern is one of the typical requirements for inconsistency management techniques described by Becker et al [16]. Some authors explicitly aim to exploit the benefits of such an approach (Egyed et al [60], Giese et al [80], Gausemeier et al [78]). While not always emphasized, incremental evaluation is usually handled as a desirable property of inconsistency management techniques.

In Table 3.4, we summarize the patterns supported by the inconsistency management techniques investigated in this paper.

	Characterization					Detection
	Representation			Specification		
	Naming conventions	Dependency graphs	Other	Human inspection	Similarity analysis	Incremental
Adourian [4]	○	●	○	●	○	○
Balzer [12]	○	○	○	●	○	○
Becker [16]	○	●	○	●	○	○
Bhave [24]	○	●	○	●	○	○
Bhave 2 [23]	○	○	● (pivot)	●	○	N/A
Blanc [27]	○	○	● (operation)	●	○	●
Dahman [40]	○	●	○	●	○	○
Easterbrook [55]	○	○	● (logic)	●	○	○
Egyed [60]	○	●	○	●	○	●
Engels [62]	○	○	● (formal)	●	○	○
Gausemeier [78]	○	●	○	●	○	●
Giese [80]	○	●	○	●	○	●
Giese 2 [79]	○	○	● (formal)	○	○	●
Hamlaoui [61]	○	●	○	●	○	●
Hehenberger [88]	○	○	● (ontology)	●	○	●
Herzig [89]	○	●	○	○	●	N/A
Hessellund [92]	●	○	○	●	○	○
Le Noire [113]	○	○	● (operation)	●	○	○
Lopez-Herrejon [118]	N/A (avoidance only)					○
Lopez-Herrejon 2 [119]	○	○	● (logic)	●	○	●
Mens [128]	○	●	○	●	○	○
Nentwich [136]	N/A (avoidance only)					○
Oka [141]	●	○	○	●	○	○
Qamar [150]	○	●	○	●	○	○
Quinton [152]	○	○	● (SAT)	●	○	○
Shah [167]	○	○	● (pivot)	●	○	○
Stolz [175]	○	○	● (operation)	●	○	●
Van Der Straeten [192]	○	○	● (logic)	●	○	○
Van Der Straeten 2 [191]	○	○	● (rules)	●	○	○

Table 3.4: Inconsistency management patterns in the state of the art. (Legend: ○ – no support; ○ – possible support, not emphasized; ● – dedicated support)

### 3.1.9 Resolution patterns

Resolution patterns address detected inconsistencies. Depending on their intent, changing and non-changing actions are distinguished [172], i.e. the ones aiming to bring the models back to a consistent state, and the ones aiming to notify stakeholders about inconsistencies and trigger further (manual) evaluation, respectively. From an automation point of view, manual and semi-automated patterns can be distinguished. (We argue that full automation cannot be achieved in general cases and human intervention - especially from the domain experts and stakeholders - is inevitable.) Additionally, Gausemeier et al [78] distinguish between operations that required user interaction and the ones that are straightforward to execute, i.e. no further inconsistency is introduced by the resolution.

#### **Model synchronization and change propagation**

Model synchronization and change propagation are commonly used patterns for automating inconsistency resolution. [78, 40, 4, 80, 136, 150, 128] As a prerequisite, appropriate detection algorithms are required to be deployed, which are capable to signal inconsistent states between the participating models. (E.g. by graph reasoning (Section 3.1.8).)

#### **Editing hints**

Editing hints are also a commonly used pattern, although requiring more human participation. During the resolution phase, the user is provided by resolution/editing hints by the modeling environment. The environment is responsible only for generating and visualizing the hints; it is the user who decides which one of the hints to apply. Hessellund et al [92] present a prototype tool that queries a Prolog fact base for valid model change operations and these are presented in a pop-up view. Hegedus et al [86] present a technique for generating quick fixes for DSLs.

#### **Design-space exploration**

Design-space exploration (DSE) is another pattern that can be used in combination with editing hints and explicitly model resolution processes. The modeling environment is integrated with an appropriately configured DSE engine, which is responsible to generate an exhaustive list of resolution strategies. An example is provided by david2016streaming et al [44].

#### **Logic-based resolution**

The logic-based representational pattern (Section 3.1.7.1) is often used in conjunction with similarly logic-based resolution patterns, such as in Van der Straeten et al [192].

### 3.1.10 Optional and auxiliary activities

As Figure 3.3 shows, apart from the mandatory activities discussed previously, there are optional activities an inconsistency management technique may incorporate. At this place, we focus on tolerance, tracking and process optimization.

### 3.1.10.1 Tolerance

Finkelstein et al [70] hint that instead of just removing an inconsistency, we have to reason about *managing* them. In our view, this also includes *tolerating* detected inconsistencies before executing potentially costly resolution actions, i.e. postponing the resolution activity as much as viable in order to allow the potential resolution of *transient* inconsistencies without intervention.

Blanc et al [27] propose to represent models by sequences of elementary construction operations. To detect structural constraints, they define pairs of operations, where the second operation cancels out first one. E.g. creating and subsequently deleting a model element. Although the authors do not investigate it, the approach provides foundations for **temporal inconsistency tolerance**. In these scenarios, inconsistencies are tolerated based on temporal and timing relations of elementary or compound model changes, or operations.

Balzer et al [12] focus on augmenting instances of inconsistencies with *state*. The authors introduce the notion of temporal tolerance by deconstructing inconsistency rules to two derived rules, the appearance and disappearance rule which span a temporal interval of the model(s) being in an inconsistent state, hence making inconsistencies stateful entities. The approach keeps track of the violated constraints by using “pollution markers” and instead of instantly initiating a resolution action, concerned processes (and stakeholders) can be notified about the inconsistency. By allowing further engineering activities to be executed during the inconsistent interval, the better parallelization of the design workflow can be achieved and ultimately, these may lead to the inconsistencies to be resolved without interrupting the design process for further reconciliation. As a limitation, the technique only deals with the most simplistic version of temporal consistency relations, in which a pair of subsequent operations form an identity transformation. In practice, more complex structures of operations have to be supported.

Easterbrook et al [55] propose a similar technique for temporal inconsistency tolerance in the context of multi-view modeling. Tolerating inconsistencies decouples the viewpoints and introduces flexibility in the design process as deciding upon *when to resolve inconsistencies* is the responsibility of the owner of the view. The authors provide a formal approach for guiding the decision in form of pairs of pre- and postconditions. Our approach extends this model by using a quantifiable distance metric to evaluate the divergence of views (and viewpoints). The distance metric also helps understanding the impact of unresolved inconsistencies and reason over their accumulation and evolution.

### 3.1.10.2 Tracking and process optimization

The management of detected inconsistency rules also may involve storing relevant information upon detection, such as the affected model element, timestamp, etc. This data can be later reused in various ways, for example

- revising inconsistency rules,
- fine-tuning the tolerance and resolution approach,

- optimizing the engineering process in order to minimize additional costs arising from inconsistencies.

These patterns, however, are not properly addressed by the state of the art.

### 3.1.10.3 Measuring inconsistency

Measuring inconsistency has been a topic of interest in the ontology and knowledge engineering. Hunter and Konieczny [94] approach measuring the level of inconsistency through the notion of minimal inconsistency sets. They show how their inconsistency metric is a special Shapley Inconsistency Value, which enables using SAT techniques for proving consistency and inconsistency. Ma et al [122] propose a technique to measure the degree of inconsistency in description logic based ontologies. As compared to these approaches, we mainly focus on collaborative modeling of complex engineered systems where the semantic inconsistencies of the models are not that obvious as in ontologies.

Inconsistency measurement approaches in software engineering settings typically focus on inconsistencies on linguistic level. Lange et al [112] use the number of detected instances of various inconsistency rules between UML diagrams as an inconsistency metric between views. Inconsistency rules are syntactic, such as messages in sequence diagrams without names.

Similar to our approach is the one presented by Barragáns-Martínez et al [13], who provide a formal framework to assess the significance of inconsistencies in requirement specifications. The scope of our work is more general as our technique is not constrained to requirement specifications but arbitrary models in a collaborative setting.

### 3.1.11 Consistency in other contexts

Consistency issues are of course well known in distributed and multiprocessor contexts, where program correctness is strongly dependent on understanding precise conditions under which the memories of different processors may differ. Lamport's seminal work on parallel computers, for instance, developed core notions of consistency as preserving *causality* between events [110], or in terms of interleaving sequential streams in the well known *Sequential Consistency* model [111]. A wide variety of less strict, or "relaxed" consistency models have since been defined, although they are often quite specific to the design decisions made in the underlying hardware [166]; Sorin et al.'s book provides a good overview [171] of memory model designs and issues.

A number of approaches to categorizing consistency have also been attempted. For distributed, virtual environments, Bouillot and Gressier-Soudan decompose consistency into elements of causality, concurrency, simultaneity, and instantaneity [29]. The latter two are difficult to achieve, and typically imply sacrificing the former two and living with short-term inconsistencies. Liu et al.'s survey paper organizes models based on their focus on *ultimate consistency* (systems which become eventually consistent), or on being *deadline-based* (given an event at time  $t$ , consistency is achieved at  $t + \delta$ ). The former includes Lamport's basic models, as well as various similar forms of *serializability* [19], while the latter can be further broken down into perceptive (or absolute) consistency [29], where every process

executes events at the same absolute time, *delayed consistency* [151], which imposes a fixed, maximum pair-wise delay of  $\delta(i, j)$  between processes  $i$  and  $j$ , and *timed consistency* [182], which requires a fixed global bound of  $\delta$  on all processes. In this terminology our definition of eventual consistency is an instance of ultimate consistency, exact consistency a kind of absolute consistency, and regular consistency is deadline-based. Repetitive consistency and bounded consistency introduce new ideas based on a flexible notion of delay, and a formal distance metric for relative consistency.

### 3.1.12 Conclusions

After investigating the broad topic of inconsistencies in multi-model mechatronic/CPS settings, we draw the conclusions in this section and answer the questions posed at the beginning of Section 3.1.

**What methods, techniques and tools are available for the management of consistency in a mechatronic and CPS design process?**

We found that there are only a few inconsistency management techniques efficiently supporting the design of CPS/mechatronic systems. These foundational works focus mainly on characterizing inconsistencies as a necessary first step towards inconsistency management. The majority of the inconsistency management techniques of the state-of-the-art have their roots in software system modeling and their applicability in CPS/mechatronic design processes is questionable.

**What are limitations of the state-of-the-art?**

*Focus on software system modeling and UML in particular.*

As mentioned earlier, the main limitation of the state-of-the-art is the focus on software system modeling and the lack of techniques to reason about physics in terms of inconsistencies. A majority of the publications considered in this survey assume UML as the de facto modeling language and implicitly tie the techniques and approaches to UML. Even though UML is supposed to be a general-purpose modeling language, its applicability in CPS/mechatronic design is marginal, as only a small set of design tasks are suitable to be efficiently addressed by UML or one of its variants (SysML, MARTE, etc). This strong relation to UML makes most of the state-of-the-art techniques unfeasible to apply in CPS/mechatronic design.

*Tolerance.*

Tolerance is an optional activity in allow-and-recover style of inconsistency management techniques, situated between detection and resolution. Its various forms (temporal, design, spatial) enable better runtime performance and better scalability of the overall design process. Inconsistency tolerance has been addressed sporadically by the state of the art and there is no comprehensive, structured approach on the topic.

*Explicitly modeled resolution processes.*

While many authors propose semi-automated human-guided resolution techniques (Hessellund et al [92], Hegedus et al [86]), the appropriate resolution actions typically require complex sequences of changes. Explicitly modeling resolution actions is a missing link in the state of the art that.

*Process optimization.*

By reusing the data on encountered (i.e. detected and resolved) inconsistencies, the overall engineering process can be optimized. For this purpose, appropriately designed databases and smart analysis techniques are required. This activity, however, is not addressed by the state of the art.

Figure 3.5 summarizes our view on efficient inconsistency management techniques in the mechatronic domain, which is based on our survey of the state of the art and extended by our suggestions.

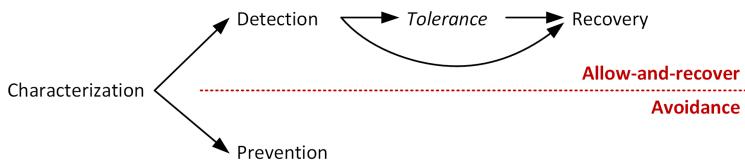


Figure 3.5: Extended view on inconsistency management styles and techniques.

**What makes consistency management in a mechatronic and CPS context different from the software engineering industry in general?**

The main difference between modeling pure software systems and mechatronic systems is the involvement of models of physics in the latter case. Incorporating physics in the design of a virtual product links the whole design process to a belief system with inherited semantics. In contrary, the semantics of a pure software system are fully to be defined. Reasoning about inherited semantics, and especially: modeling the behavior of inherited semantics is what makes mechatronic design radically different from software design. The involvement of physics is also the reason of many software- and UML-oriented inconsistency management techniques being unfeasible to apply in mechatronics.

## 3.2 Process engineering

In this section, we review the state of the art on process modeling formalism and process performance assessment techniques.

### 3.2.1 Process modeling formalisms

Process modeling languages are primarily geared towards modeling concurrency and synchronisation [187]. Pertinent examples include languages based on the *Business Process Model and Notation* [174], *Petri nets* [185] and *UML Activity Diagrams* [22].

The *Business Process Model and Notation* (BPMN) [169] is a widely used standard in process modeling. BPMN is used in a wide range of areas, to model processes in non-IT, as well as IT-intensive organisations. Its main goal is to provide an understandable notation for all stakeholders. The focus is more on the conceptual modeling of processes, and less on orchestration and execution. In version 2.0, the standard has been extended with support for orchestration, albeit on a non-technical level.

*jBPM* [39] is an open-source, Java-based framework that supports execution of BPMN 2.0 conform processes. The framework also provides enhanced integration features with external services in the form of managed Java program snippets. In addition, the process engine is tightly integrated with a collaboration and management service (Guvnor), a standardized human-task interface (WS-HT), a rule engine (Drools) and a complex event processing engine (Drools Fusion).

The *Business Process Execution Language* (BPEL) [210] is a standardised language for specifying activities by means of web services. The standard specifies a BPEL process as XML code, though graphical notations exist, often based on BPMN. Service interaction can be executable or left abstract. Analysis tools for BPEL have been developed, for example by formalising BPEL models in terms of *Petri nets* as done by Ouyang et al [144] and Xia et al [213]. Kovács et al[106] use a symbolic analysis model checker. Fu at al [76] and Foster et al [73] analyse the communication between BPEL processes by employing automata. Nevertheless, BPEL is exclusively used for web services defined using WSDL.

*Yet Another Workflow Language* (YAWL) [186] attempts to combine the functionality of BPMN (business-mindedness) and BPEL (executability). In contrast to other approaches, YAWL was designed with formal semantics in mind, and is defined as a mapping to *Petri nets*. Execution particularly aims to provide insight in data and resources. There is, however, no particular focus on the integration and orchestration of tools.

*Orc* [104] is a formal textual language for the orchestration of service invocation in concurrent processes. It aims to manage timeouts, task priorities, and failure of services and communication. Orc is based on trace semantics, which is used to determine whether two Orc programs are interchangeable. The integration of tools can be achieved by defining sites, which represent units of computation. There is no support, however, for modeling modal behaviour, and the textual notation does not scale to large processes.

*Open Services for Life-cycle Collaboration* (OSLC) [142] is the de facto standard in tool integration. It is a specification for the management of software lifecycle models and data, which are represented as resources. The specification is intended to be used for integration of services and data, and does not include process modeling.

The *Statecharts* formalism [85] has first-class notions of concurrency, hierarchy, time and communication. It can therefore be viewed as a suitable formalism for integration and orchestration. Because *Statecharts* is state-based, and does not include *fork* and *join* constructs, it is less suitable for process modeling. *Statecharts* has been combined with

*Class Diagrams* in SCCD [196], to provide structural object-oriented language constructs (*i.e.*, objects with behaviour).

*Story diagrams* [206] are a formal behavioral specification language with process semantics. Similarly to UML Activity Diagrams, they describe control and data flow across the process, but with the added support for specifying executable actions. Just like the FTG+PM formalism, story diagrams rely on typed attributed graph transformations, but with a very simplistic type model. As a result, even though story diagrams provide added behavioral specification semantics, the formalism still is not as versatile as the FTG+PM.

A summary of all approaches and their suitability for our purpose is presented in Table 3.5. We have investigated whether the approach is intended to be used to specify processes (**process**), whether it aims at integration of services/tools (**integration**), whether it supports execution or enactment (**executability**), whether it provides means for formal analysis (**analyzability**), and whether its notation is appropriate for the tasks it is intended for (**usability**).

	Process	Integration	Executability	Analyzability	Usability
Approaches	●	○	●	●	○
	●	○	●	●	●
	●	○	●	●	●
	●	●	●	○	●
	●	○	●	●	○
	●	○	●	●	●
	●	●	●	●	○
	●	●	●	●	●
	●	○	●	●	●
	○	●	●	○	●
SCCD					
Story diagrams					

Table 3.5: Summary of the related work. (● – Supports, ○ – Partially supports, ○ – Does not support)

The main conclusion is that no approach truly unifies process modeling and integration of services. The approaches that score best in these two aspects (BPEL and Orc) do not have an intuitive, accessible notation, although in the case of BPEL, graphical notations have been suggested but are not part of the standard. jBPM overcomes these shortcomings, but does not support analyzability of the service interactions.

### 3.2.2 Performance analysis techniques

Performance analysis of processes constitutes an important part of our work. Due to the semantic variety of different process/workflow formalisms, however, there is no single solution for the performance evaluation of process models in general.

Alshareef et al [7] map processes to parallel DEVS models with the intent of behavior specification. The authors base themselves on UML Activity diagrams. Unfortunately, UML Activity diagrams do not include all of the *van der Aalst* patterns [187], which

is paramount to the current thesis. Additionally, our intent of mapping to DEVS is the performance evaluation of stochastic processes in combination with resources.

The idea of using DEVS for performance evaluation already popped up in the work of Cohen [36]. The authors view discrete event systems as linear algebras, in which the periodical behavior (i.e., repeatedly performed activities) can be characterized by solving an eigenvalue and eigenvector equation. Numeric algorithms are used to solve the equations and link the results to performance metrics of manufacturing systems.

The linchpin of assessing process performance is the calibration of the process as realistically as possible. Automating this step is a big step towards realistic simulation results. The most natural way to do so is by processing data from the previous runs of the process. Li et al [116] extend the WF-net formalism with timing information and provide a formal framework for assessing the lower bound of average turnaround time of the process. The timing information, however, is not modeled by statistical distributions, but scalar metrics. Xiao et al [214] focus more on the problem of the competition for the limited resources within the process. They propose a method based on queuing theory in order to analyze the time performance of a process. Miu et al [131] use learning algorithms over historical data to predict execution times of single activities based on the characteristics (instances and attributes) of their input data.

The resource constrained project scheduling problem (RCPSP) [9] is a dual problem to process performance optimization. Its focus lays on the schedule of the activities that has to be determined in the presence of resource constraints and timing information.



# Chapter 4

## Correctness and consistency

In this section, we present **Contribution 2** of this thesis: adapting the definition of (in)consistency to match our needs for the management of semantic inconsistencies.

In this section, we present the core problem this thesis is concerned with: managing model inconsistency in order to retain the eventual correctness of the product. We base our reasoning on the fact that delivering the *correct* product at the end of the engineering process is a must. We will argue that carrying out the engineering process *consistently* helps achieving the correct product and increases the efficiency of the process.

As outlined in Section 2.1, requirements are used to obtain the properties the final product must satisfy. From this point on, we assume an appropriate mapping from requirements to the properties and approach the problem of (in)consistency management in terms of properties only. To do so, we will use the concepts shown in Figure 4.1

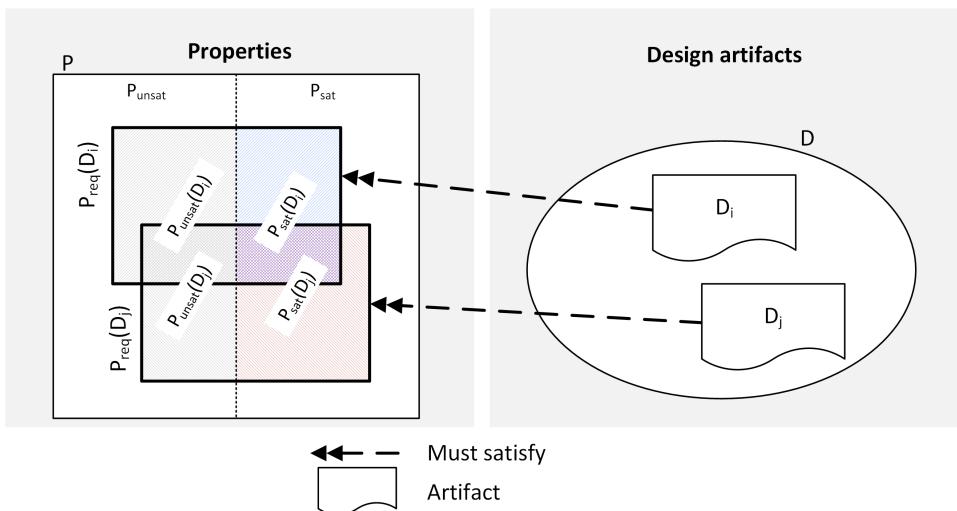


Figure 4.1: The relationship between properties and designs.

The set of system properties  $P$  can be divided as follows.  $P_{req}(D_i) \in P$  denotes the subset of properties required to be satisfied by a design artifact  $D_i$ . The design artifact may or may not satisfy these properties, denoted by  $P_{sat}(D_i)$  and  $P_{unsat}(D_i)$ , respectively. Formally, the following hold:

- $\forall p \in P_{sat}(D_i) : D_i \models p;$
- $\forall p \in P_{unsat}(D_i) : D_i \not\models p;$
- $P_{sat}(D_i) \cup P_{unsat}(D_i) \equiv P_{req}(D_i).$

Given another design  $D_j$  with the same respective set of properties  $P_{sat}(D_j)$  and  $P_{unsat}(D_j)$ , we are able to formalize correctness and consistency.

## 4.1 Correctness

The (virtual or real) product may or may not meet the requirements. In this thesis we consider the product to be correct if and only if it meets all the requirements. If at least one requirement is not met, the product is considered an incorrect product.

Any incorrectness introduced during the realization of the real product from its virtual representation is considered to be out of the scope of this thesis. Therefore, the correctness of the product is approximated with the correctness of the virtual product.

### Correctness of a design.

Design  $D_i$  is said to be correct with respect to a set of required properties  $P_{req}(D_i)$  iff  $\forall p \in P_{req}(D_i) : D_i \models p$ .

As a consequence, there must be no unsatisfied required property:  $P_{unsat}(D_i) \equiv \emptyset$ .

The correctness of the overall virtual product is given by the correctness of the design artifacts.

### Correctness of a virtual product.

The virtual product  $D \equiv \bigcup_{D_i \subseteq D} D_i$ , is said to be correct with respect to  $P$  iff  $\forall p \in P \exists D_i \in D : D_i \models p$ .

## 4.2 The repercussions of ensuring correctness

Retaining the correctness of the product has its own repercussions on the engineering endeavor. This is because retaining correctness and the costs of the engineering do not trivially align well. The problem is, ensuring correctness at every elementary engineering step slows down the engineering work tremendously. Overlooking correctness, however, could result in an incorrect product. Noticing incorrectness too late results in exponentially increasing costs and necessary re-iteration in the engineering work.

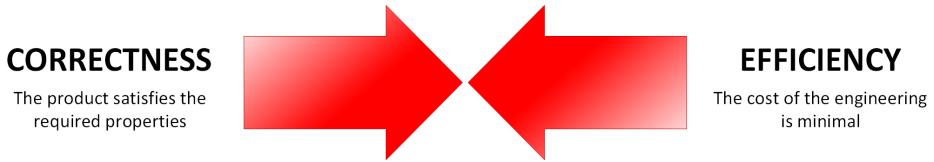


Figure 4.2: Correctness often contradicts efficiency.

In real heterogeneous settings some activities aiming to ensure correctness may take weeks or even months. This is typical in prototype test scenarios, e.g. when a scaled model of an airplane is built and used in a wind tunnel to observe its aerodynamical properties.

A trade-off between checking correctness too often and rigorously, and checking correctness only in the integration phase has to be made.

### 4.3 A heuristic for eventual correctness

We often think in terms of *heuristics* when a goal is to be met, but the set of practical solutions are not guaranteed to lead to the goal.

**Heuristic (Romanycia and Pelletier [157]).** Any device, be it a program, rule, piece of knowledge, etc., which one is not entirely confident will be useful in providing a practical solution, but which one has reason to believe will be useful, and which is added to a problem-solving system in expectation that on average the performance will improve.

Motivated by collaborative modeling and concurrent engineering settings, in this thesis, we view **model consistency as a primary heuristic** for the eventual correctness of the (virtual) product. Keeping models consistent with each other is a significantly easier task as compared to keep models correct.

The above definition is then adopted as follows. The *device* is the consistency between the design artifacts, whose presence does not *entirely confidently* provides the *practical solution* (correct product). This heuristic is added to the *problem-solving system* (the engineering endeavor) in the expectation on that the *average performance will improve* (the number of attempts required to come up with the correct product will decrease).

Looking at model consistency as a heuristic suggests, that *the belief* that more consistent designs will more probably lead to a correct product is indeed plausible. At the same time, this notion also warns us about consistent design *do not guarantee* a correct product.

One should, therefore, handle inconsistencies as the necessary part of the everyday engineering endeavour, embrace them and, by paraphrasing Finkelstein [70]: instead of removing them immediately, one should think whether or not a given instance of an inconsistency can be tolerated for a while.

### 4.3.1 Consistency

In our terms, two artifacts are consistent with respect to property  $p$  iff their respective satisfaction relation towards  $p$  is exactly the same. Following the notation of Figure 4.1, we define the consistency between two designs with respect to their *overlapping* required properties.

**Consistency of two design artifacts.**

The design artifacts  $D_i$  and  $D_j$  are said to be consistent w.r.t to the set of properties  $P' \equiv P_{req}(D_i) \cap P_{req}(D_j)$  iff  $\forall p \in P' : D_i \models p \Leftrightarrow D_j \models p$ .

That is, when deciding upon the consistency of two design artifacts, we demand the two artifacts having a respective set of required properties, which *overlap*. ( $P' \equiv P_{req}(D_i) \cap P_{req}(D_j)$ ) To the overlapping properties must be satisfied by both of the artifacts. An inconsistency arises when exactly one of the two artifacts satisfies the property. (But not the other one.) If both artifacts satisfy the property, or neither of them does, the artifacts are considered consistent.

### 4.3.2 The relationship between (in)correctness and (in)consistency

While correctness and consistency correlate, they don't imply each other. Table 4.1 shows how the satisfaction of the required properties by two respective design artifacts  $D_i$  and  $D_j$  lead to (in)correctness and (in)consistency.

Table 4.1: Consistency does not imply correctness.

	$D_i \models P'$	$D_j \models P'$	Consistent	Correct
(1)	✓	✓	✓	?
(2)	✓	–	–	–
(3)	–	✓	–	–
(4)	–	–	✓	–

$P'$  denotes the overlap between the two sets of required properties for the two respective design artifacts, i.e.  $P' \equiv P_{req}(D_i) \cap P_{req}(D_j)$ .

It is obvious that if both  $D_i$  and  $D_j$  meet the shared set of requirements, they are consistent. This, however, does not say anything about  $D_i$  and  $D_j$  meeting the rest of their respective requirements, and therefore, their correctness cannot be inferred.

Should either  $D_i$  or  $D_j$  not meet the shared requirements, they are definitely inconsistent w.r.t the properties of the shared requirements; and the resulting product will be incorrect, due to at least one of the models not satisfying the required properties.

When neither  $D_i$  nor  $D_j$  meet the requirements, they can be still considered consistent, but they will be *consistently incorrect* way. In this case, even though the models seem

not be inconsistent, at the end of the development process, the resulting product will be incorrect.

The following conclusions can be drawn from Table 4.1.

Consistency is a required, but not satisfactory requirement to correctness. That is, to build the correct product, the designs *eventually* have to be consistent with each other.

Formally:

- $\text{consistent}(D_i, D_j) \neq \text{correct}(D);$
- $\text{correct}(D) \Rightarrow \text{consistent}(D_i, D_j).$

Conversely:

- $\text{inconsistent}(D_i, D_j) \Rightarrow \text{incorrect}(D);$
- $\text{incorrect}(D) \neq \text{inconsistent}(D_i, D_j).$

## 4.4 The need for explicitly modeled processes

Now, that we established, that managing inconsistencies helps delivering the correct product, we should focus on a formal framework for reasoning about *when* and *how* to manage inconsistent models.

We look at the task of inconsistency management as a typically efficient and useful approach for ensuring eventual correctness. Still, wrongly executed inconsistency management may have severe repercussions on the time it takes to produce a correct product. For example, dealing with incompatible sub-system interfaces during the integration phase may require additional iterations over costly engineering activities. Consequently, analyzing the impact of various inconsistency management techniques is desirable.

The explicitly modeling the engineering process, the various inconsistency patterns and management patterns, enables the formal reasoning about the optimal inconsistency management strategy, pertaining to the specific engineering endeavor.

By explicitly modeling the engineering process, combining the two facets of proper and efficient inconsistency management can be achieved: the *when?* and *how?* can be posed as well-formalized problems. Additionally, the process model enables thorough understanding of where and when specific inconsistencies arise. On the other hand, it also enables a quantitative assessment of the impact of introducing a specific inconsistency management strategy to the process. This concept is shown in Figure 4.3.

In the later sections of this thesis, we will present a framework for modeling and enacting processes, while also managing inconsistencies in them. Standard process modeling formalisms, however, fall short of capturing an important aspects of the engineering process that enables reasoning over consistency: the notion of *semantic properties*.

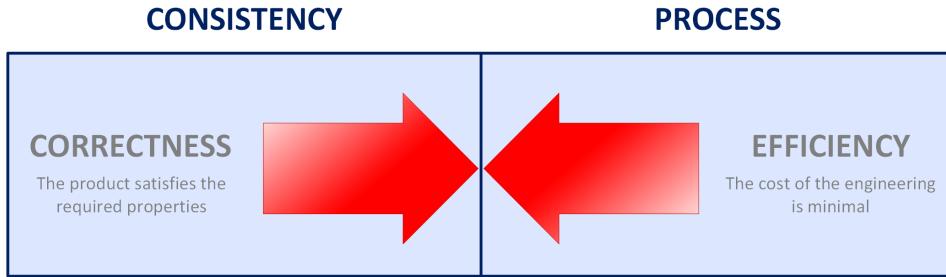


Figure 4.3: Correctness is approximated by consistency; efficiency is measured in terms of the process.

To reason about inconsistencies and their management, the process modeling formalism should be capable of depicting (semantic) properties of the system and relate them to artifacts and activities of the process.

In Section 5.1 we present our formalism that satisfies this requirement. But first, we should investigate how the properties of the system are obtained and how do they relate to the engineering itself.

## 4.5 System properties for reasoning over consistency

The *properties* of the system play an important role in formalizing inconsistencies.

Benveniste et al [17] use three operators to describe the activities of engineers in contract-based engineering settings. Using this as a baseline, we make the assumption that every engineering process consists of activities of *five* types, as shown in Figure 4.4.

The engineering process commences by understanding and formalizing requirements, as discussed in Section 2.1.1. The output of this activity is the set of properties the eventual design has to satisfy.

The subsequent engineering activities can be viewed as one of the five types of activities shown in the figure:

- architectural decomposition,
- view decomposition,
- refinement followed by abstraction,
- abstraction followed by refinement, and
- the elementary engineering operation.

Every activity takes the set of the properties  $P$  (or a subset of it), and produces a design  $d$ . An activity may also use a design as an input to execute the engineering operation on that design. Conceptually, this is equivalent with the process in Figure 4.4.

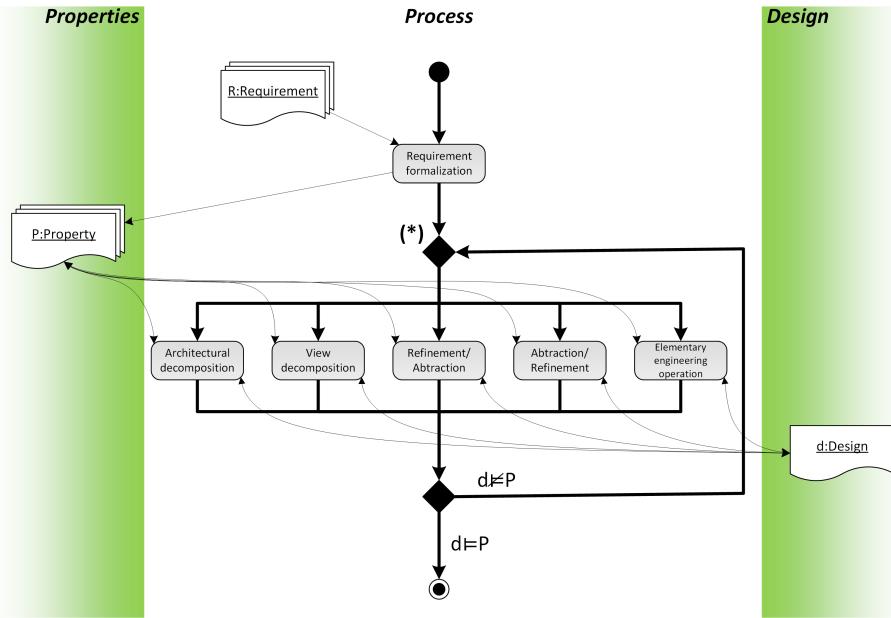


Figure 4.4: The five types of engineering activities.

Following the chosen activity, it is determined if the properties are satisfied ( $d \models P$ ) or not ( $d \not\models P$ ). In the former case, the process terminates successfully; in the latter case, however, another engineering activity has to be taken.

Additionally, the nesting of activities is also allowed and this is what is typical in real engineering processes. We show this mechanism in details in the following.

### 4.5.1 Architectural decomposition

The architectural decomposition activity (Figure 4.5) decomposes the system into architectural components. This activity is usually situated on a higher level of the engineering. It is very often encountered, for example, in OEM company processes, where that parallel engineering processes are be facilitated, *without* having any impact on each other.

It is important to note, that the *architecture*, ideally, is yet another design artifact in the engineering process.

The main focus are the *interfaces* and their compatibility. In the *System integration* part, the interfaces are the primary means of re-composing the higher level architectures.

Between the *Architectural decomposition* and the *System integration* activities, the engineering process has the possibility to re-iterate to the segment of the process in Figure 4.4, marked by  $(*)$ . This mechanism enables *nesting* various activities into each other.

This sub-process also appears in the work of Benveniste et al [17], as part of the contract composition and system integration scenarios.

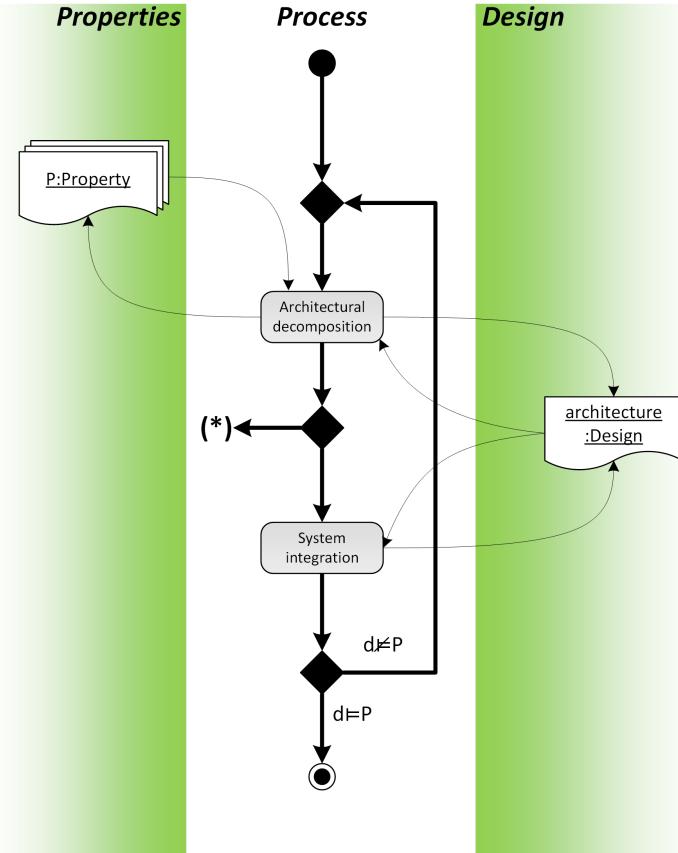


Figure 4.5: The sub-process of *Architectural decomposition* and *System integration*.

### 4.5.2 View decomposition

The view decomposition activity is the one that facilitates the multi-view and multi-paradigm nature of complex engineering processes. View decomposition should follow the guidelines outlined in the related ISO standard 42010 [102].

A viewpoints capture stakeholders interests and concerns; concrete languages and tools; and abstract formalisms. [32]. Applying a viewpoint to a specific problem results in a view.

From our perspective, the decomposition the system into various views is a correspondence function which projects a subset of properties:  $\text{decompose}_v(\text{sys}) : v \mapsto P' \subseteq P$ . This is referred in the ISO standard as “the view being the result of applying the viewpoint to a particular system”.

As a by-product, the formalisms used in  $v$  are also determined in this step, as discussed in the ISO definition. Thus, it is safe to assume the most appropriate formalisms are used to reason about properties  $P'$ . This is the step which eventually results in overlapping sets of properties over various views [148] and gives rise of inconsistencies as the engineering

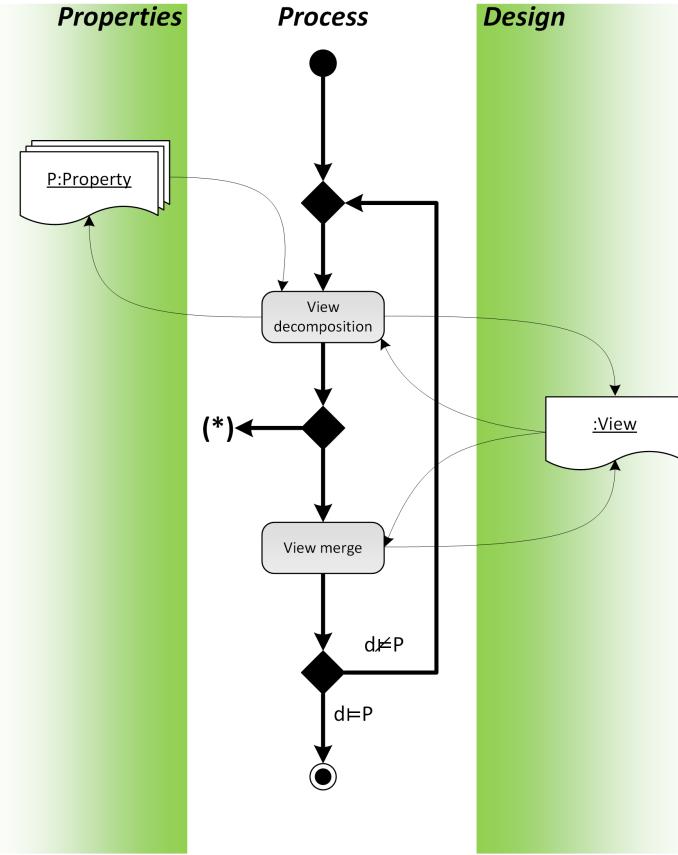


Figure 4.6: The sub-process of *View decomposition* and *View merge*.

process is advances. Our approach is, however, independent from the details of certain views, as long as the properties are appropriately modeled in our formalism.

Merging separate views is often required for simulating the system at some point of the engineering process. The merge is accomplished by the union operation of properties:  $v' \oplus v''$ . Given  $v' \models P' = \{p'_1, p'_2 \dots p'_M\}$  and  $v'' \models P'' = \{p''_1, p''_2 \dots p''_N\}$ , it must hold that  $v' \oplus v'' \models P' \cup P''$ , i.e.  $v' \oplus v'' \models p'_1 \wedge p'_2 \wedge \dots \wedge p'_M \wedge p''_1 \wedge p''_2 \wedge \dots \wedge p''_N$ . This step is used when two views have to be jointly investigated, e.g. for co-simulation purposes.

### 4.5.3 Refinement-abstraction

Refinement and abstraction are the most natural steps in not just engineering processes, but also in the very fundamental cognitive processes.

The *Refinement* activity takes a design  $d$  with its related properties  $P$ , and produces another design  $d'$  and its related properties  $P'$ . The activity is a refinement iff  $P \subseteq P'$ , i.e. the set

of concerned properties grows. The refinement is valid iff  $\forall p \in P : d \models p \Rightarrow d' \models p$ , i.e. the properties originally satisfied by the design  $d$  must be satisfied by the design  $d'$ .

Persson et al [148] define the mechanism of abstraction in terms of views and their semantic domains. According to authors, a view  $V_1$  is an abstraction of a view  $V_2$  if its semantics is a superset of the other, i.e.  $S(V_1) \supseteq S(V_2)$ .

Selic [164] provides a catalogue of abstraction patterns for model-based software engineering.

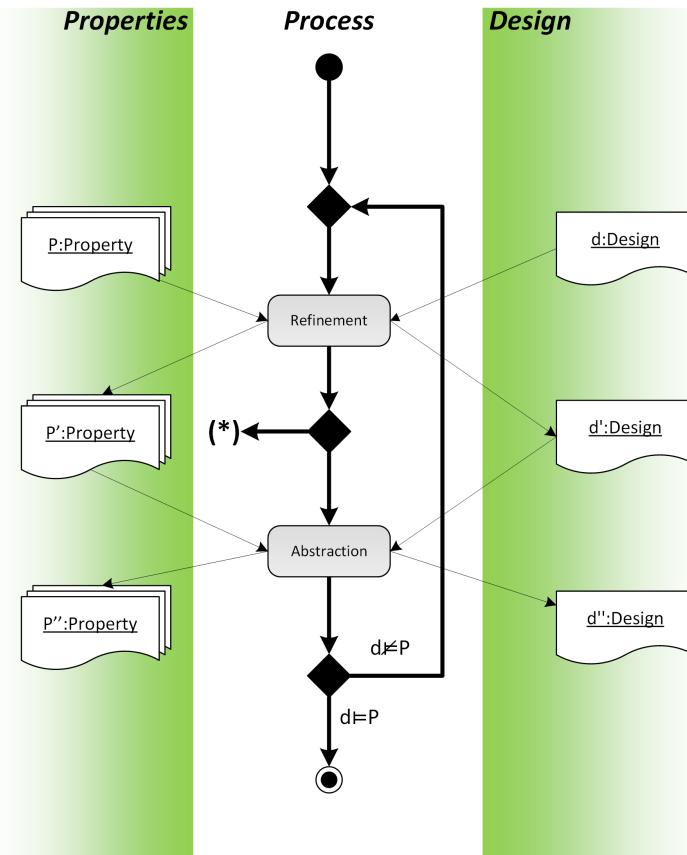


Figure 4.7: The sub-process of *Refinement-Abstraction*.

#### 4.5.4 Abstraction-refinement

As the inverted direction of the refinement-abstraction mechanism, the abstraction-refinement mechanism first abstracts a model and then refines it again. It is not as frequently encountered as the opposite mechanism, due to the fact that abstraction loses information and it is not guaranteed, that the original system can be reconstructed after having been abstracted. Nevertheless, multi-paradigm settings can benefit from such a mechanism, e.g. when

mapping to a common abstract formalism and then refining the resulting traces into more detailed models.

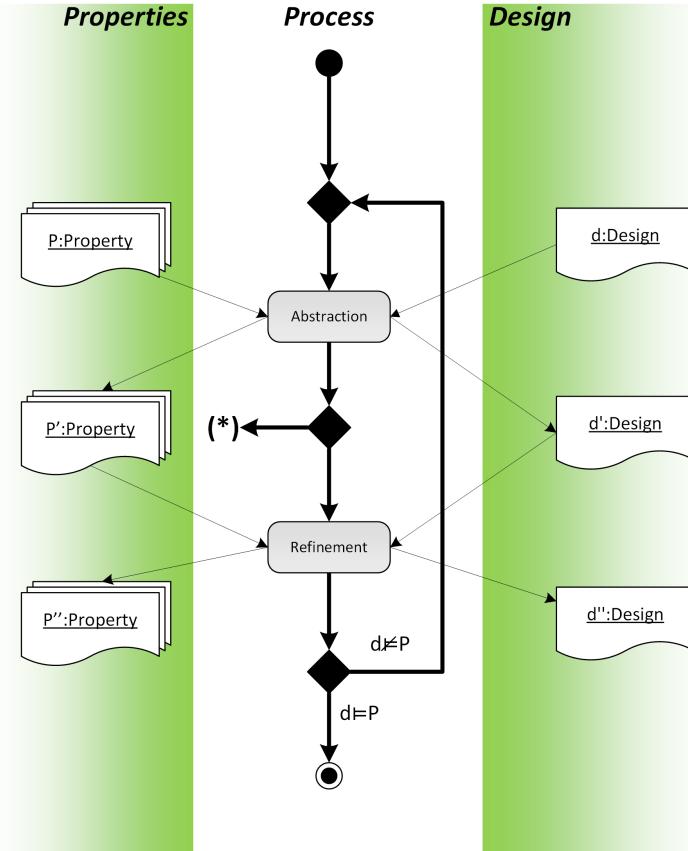


Figure 4.8: The sub-process of *Abstraction-Refinement*.

#### 4.5.5 Elementary engineering operation

The elementary engineering operation is a wildcard that represents *any* activity in an engineering process. We use it to make our view on engineering processes full.

In general, the elementary engineering operation takes a design  $d$  and produces another design  $d'$  with the intention of improving the design, w.r.t. the requirements previously set against the engineered system.

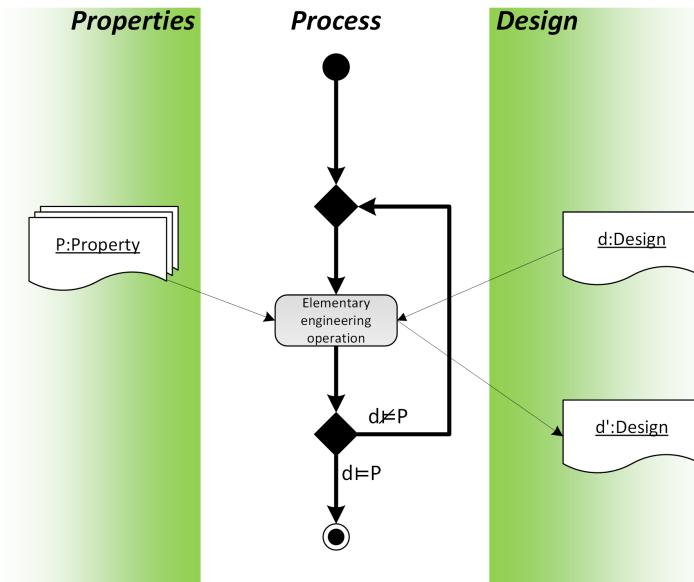


Figure 4.9: The sub-process of the *Elementary engineering operation*.

# **Chapter 5**

## **Process-oriented inconsistency management**

This chapter constitutes the core of the thesis and presents the inconsistency management framework and methodology developed during the four years of this research. In this chapter, we discuss contributions 3-8 with a strong theoretical underpinning; along with one conceptual contribution which has not been researched to its entirety due to the depth of the topic.

### **A simple case study**

We use various elements of a case study of the design of an automated guided vehicle (AGV) in this chapter.[20]. The AGV is designed to transport payload on a specific trajectory between a set of locations. The drivetrain is fully electrical, using a battery for energy storage and two electric motors driving two wheels. Being a complex mechatronic system, the requirements of the AGV are specified by stakeholders of the different involved domains, such as (i) mechanical requirements: sufficient room on the vehicle to place payload; (ii) control requirements: following the defined trajectory with a given maximal tracking error; (iii) electrical requirements: autonomous behavior, defined as the number of times that it needs to be able to perform the movement before needing to recharge; (iv) product quality requirements: the previous requirements should be achieved at a minimal cost.

Figure 5.1 shows the conceptual geometric design of the AGV. The design team chose a circular platform, with two omniwheels in addition to the two driven wheels.

The design process needs to determine the sizing of the different components (motors, battery, platform) and tune the controller. This process is decomposed into multiple dependent design steps, such as motor selection, battery selection, platform-, controller-, and drivetrain design. The process requires an interplay between different domain-specific engineering tools, such as CAD tools for platform design, Simulink and Virtual.Lab Motion for multi-body simulations employed during controller design, AMESim for multi-physical simulations during drivetrain design. Motor and battery selection activities use databases

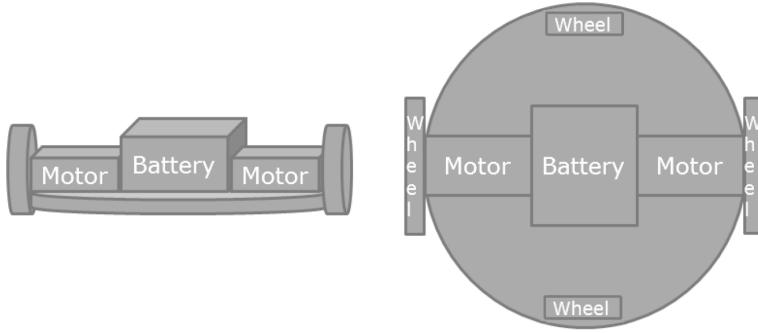


Figure 5.1: Front and top view of the conceptual design of the AGV.

maintained in Excel files. Since these tools work with different modeling formalisms, reasoning over the consistency of the system as a whole properties poses a complex problem to overcome. By explicitly modeling linguistic and ontological properties and associating them with the engineering activities, patterns of inconsistencies can be identified and handled.

### Running example

As a running example, we use a portion of the AGV process, shown in Figure 5.2. The example highlights how inconsistencies can occur due to properties of the system that interact with activities of an engineering process.

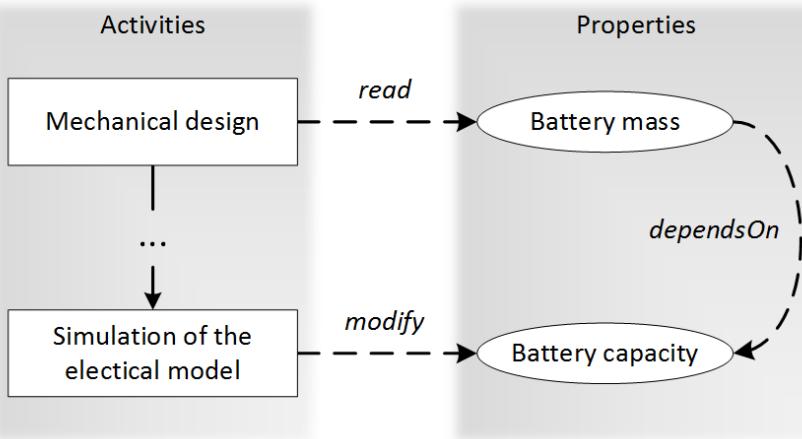


Figure 5.2: Running example.

Initially, components of the system, such as the battery, are selected based on approximations and domain expertise. The *mass* of the initially selected battery is considered during the *Mechanical design* phase to identify the mass constraints on other parts of the system. After the mechanical design phase, the electrical model is designed in details. This includes

identifying the required *capacity of the battery* by *Simulating the electrical model*, in order to fulfill the autonomy requirement.

Inconsistencies may arise when the *Battery capacity* property is changed, because the *Battery mass* property *depends on* it: batteries with bigger capacity are typically heavier. As the capacity is changed, the mass becomes inconsistent with the capacity. Should the inconsistency get unnoticed, the engineered system will fail to meet the requirements.

There are two important specificities to this example.

First, inconsistencies occur due to the lack of explicitly modeled information about *how* activities access system properties. The key in identifying the above inconsistency is the explicit modeling of the nature of interaction between activities and properties, such as *reading* or *modifying* a property. We refer to this information as *intents* of activities over (a set of) properties.

Second, state-of-the-art techniques typically reason about inconsistencies in terms of *linguistic* model elements. The dependency between the two properties is, however, not persisted in any of the engineering models as it is an inter-domain relationship. To tackle this problem, we allow modeling *ontological* properties as well, and linking them to activities by intents.

## 5.1 A formalism for modeling engineering processes

In this section, we present **Contribution 3** of this thesis: a formalism for modeling complex engineering processes. The majority of this section has been published in [41] and [42].

To model engineering processes with sufficient semantics for managing inconsistencies, we propose a formalism that augments the process with the syntactic and semantic *properties* that depict specificities of the engineered system.

We build our formalism on the FTG+PM[121] formalism, which enables the usage of process models (“PM”) in conjunction with the model of formalisms and transformations (the formalism-transformation graph - “FTG”) used throughout the process. As shown in Figure 5.3, formalisms and transformations serve as a type system to the processes: artifacts of the process are typed by the formalisms of the FTG; and activities of the process are typed by the transformations.



Figure 5.3: Relationships between processes, formalisms and properties.

Additionally, we extend the FTG+PM formalism by allowing explicit modeling of properties. We assume activities of an engineering process have a meaningful purpose of enhancing

the system. This purpose is expressed as the *intent* of an activity with respect to a property or a set of properties. Furthermore, we extend the process model by enabling specification of *costs* of single activities.

In the following, we elaborate on the specific parts of this process modeling formalism. First, we present the foundations of the FTG+PM formalism for typed processes; then we discuss the property model in details; finally we extend processes with costs.

### 5.1.1 Typed processes in the classic FTG+PM

#### 5.1.1.1 The process model (PM)

By a process  $p \in P$ , we mean a partially ordered set of activities  $A$ . The process is captured by a 4-tuple  $(A, \Delta_c, O, \Delta_o)$  consisting of

- a set of activities ( $A$ );
- a set of directed control relations between activities ( $\delta_c \in \Delta_c : A \mapsto A$ );
- a set of artifacts ( $D$ );
- a set of directed data flow relations between activities and artifacts ( $\delta_d \in \Delta_d : A \mapsto D$ );

#### 5.1.1.2 Transitivity of the control flow

The control flow is a transitive relation. That is:  $\forall a_1, a_2, a_3 \in A : \delta_c(a_1, a_2) \wedge \delta_c(a_2, a_3) \Rightarrow \delta_c(a_1, a_3)$ .

#### 5.1.1.3 Transitivity closure of an activity

The transitive closure  $\Delta_c^+$  of an activity is defined as the set of all activities reachable through the control relations  $\Delta_c$  from the activity. The following notation is used  $a_3 \in \Delta_c^+(a_1)$ .

#### 5.1.1.4 The formalism-transformation graph (FTG)

The FTG consists of formalisms and transformations, formally:  $FTG = (\mathcal{F}, \mathcal{T})$ . A transformation takes a set of formalisms and produces a set of formalisms:  $t \in \mathcal{T} : F' \subseteq \mathcal{F} \mapsto F'' \subseteq \mathcal{F}$ .

The FTG serves as a type system to processes, where languages type process objects and transformations serve as specifications to activities.

- $\forall d \in D_p \exists f \in \mathcal{F} : d \text{ instance of } f$ .
- $\forall a \in A_p \exists t \in \mathcal{T} : a \text{ realizes } t$ .

In the case study, for example, the models of the mechanical design activity are typed by a *CAD formalism*, while the activity itself realizes the transformation(s) required to achieve the engineering goals during the mechanical design, such as dimensioning the platform of the AGV, and obtaining and executing a finite element simulation model. That is,  $model_{Mech} \in D$ , and  $typeOf(model_{Mech}) = CAD$ , where  $CAD \in \mathcal{F}$ ; additionally,  $typeOf(a_{MechDesign}) \in \mathcal{T}$ .

### 5.1.1.5 Modeling support

Figure 5.4 shows the relevant excerpt of the FTG+PM modeling part of the whole formalism. The full, detailed metamodel is shown in Figure A.2 of Appendix A.

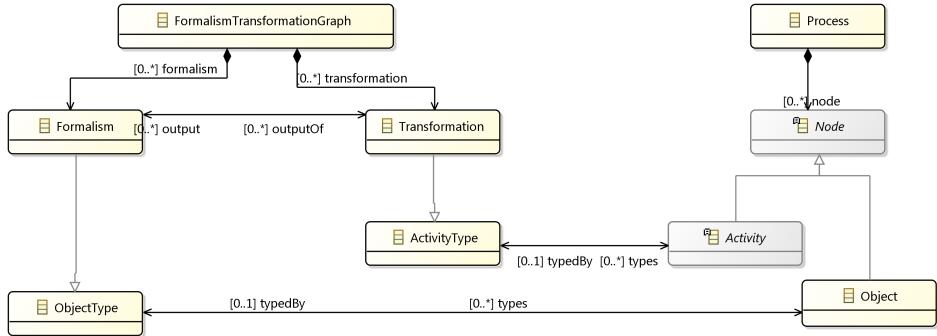


Figure 5.4: Excerpt of the FTG+PM part of the modeling formalism.

### 5.1.1.6 Modeling the running example

The modeling of the running example is demonstrated by using our prototype tool with process modeling capabilities. The tool is built upon cutting-edge Eclipse technologies. The modeling interface was built using the Sirius framework [59]. The tooling is discussed in details in Chapter 7.

**Modeling the PM** The PM of the example is shown in Figure 5.5.

This simple PM consists of two activities and two models. *mechanicalDesign* is a manual activity, as shown by the grey roundtangle and the hand icon in the upper-left corner. The activity produces a model called *design*. This model is subsequently taken by the *electricalSimulation* activity. This activity is an automated one, as shown by the yellow roundtangle and the cogwheel icon in the upper-left corner. The activity produces a model, called *elSimTrace*, which is a trace of the simulation.

The control flow between the control nodes and activities is designated by the bold black arrows. The data flow is designated by the thin grey arrows.

**Modeling the FTG** The process still has to be typed by the FTG. Generally, an ever growing FTG is assumed in MPM settings, such as the Modelverse concept, introduced by Van Tendeloo et al [200]. The FTG for the process is shown in Figure 5.6.

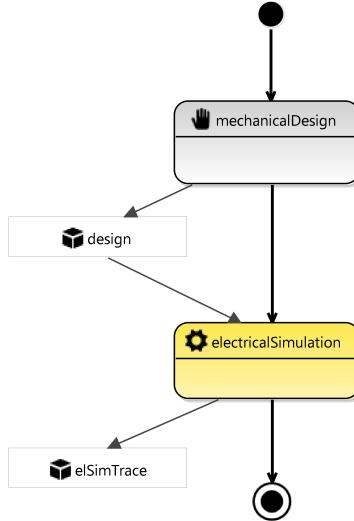


Figure 5.5: The PM of the running example.

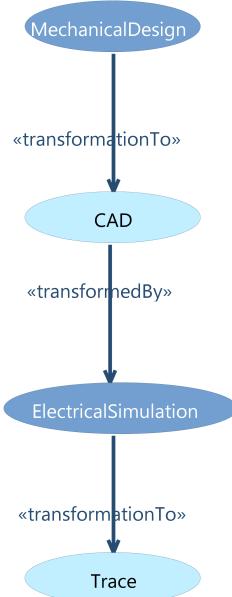


Figure 5.6: The FTG of the running example.

There are two formalisms in this simple example. *CAD* is the formalism for capturing the mechanical design. *Trace* is the formalism for capturing the results of the electrical simulation step. In between these are the transformations.

**Putting it all together** To put the two parts together, the typing relationships between the appropriate elements of the PM and the FTG are captured. The full FTG+PM is shown in Figure 5.7.

This FTG+PM is obviously a very abstract depiction of the actual engineering process and it is not suitable for any sort of analysis. For the full FTG+PM of the example, see the appendix.

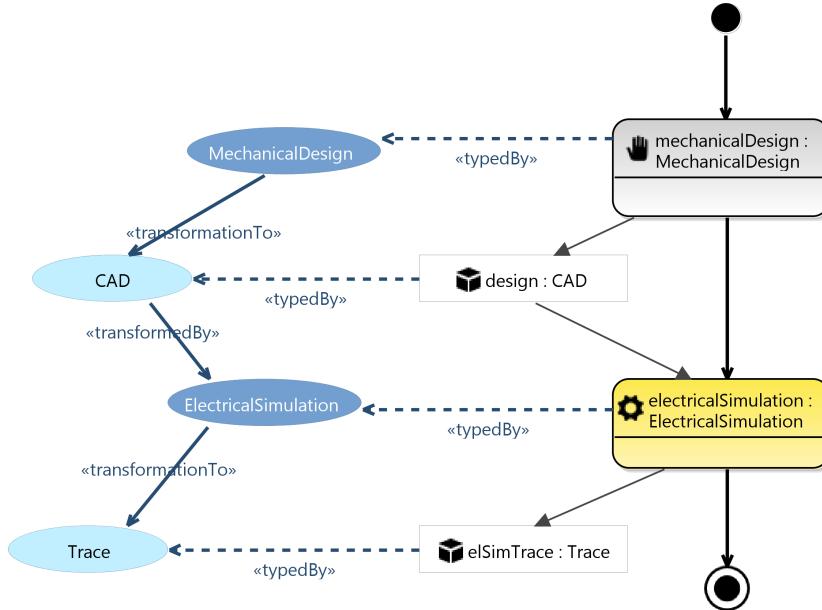


Figure 5.7: The FTG+PM of the running example.

### 5.1.2 Attributes, properties, constraints

To support the modeling of system properties, we extend the classic FTG+PM formalism by new elements: attributes, properties and constraints, that constitute the property model.

By a property model *PROP* we mean a tuple  $(\Theta, \Pi, R)$  consisting of

- the set of attributes  $\Theta$ ; and
- the set of properties  $\Pi$ ; and
- the set of influence relationships  $R$  between attributes, between properties, or between the two.

Attributes of the running example in Figure 5.2 are the *Battery mass* and *Battery capacity*, while the dependency between those is a relationship with a *direction* and a *level of precision*.

### 5.1.2.1 Attributes and properties

Attributes capture qualitative and quantitative characteristics of the modeled system and typically represent *values* of various types.

Properties capture system characteristics in terms of *satisfaction* constraints. The way properties are *checked*, also depends on their types.

Checking an attribute means evaluating if the actual value of the property is within a range of acceptance criteria; checking a property, on the other hand, means evaluating if the property is satisfied or not. While the former check is typically achieved by well-defined operators over the algebraic structures of the type of the property (e.g. arithmetic operators over number values), the latter type of checks typically involves simulations or model checking tasks. [203]

Explicitly modeling attributes, properties and their relationships (i) enables reasoning over these specificities, and (ii) fosters communication of tacit knowledge, which is especially important in the early phases of a multidisciplinary design process [203]. In our approach, attributes and properties are treated uniformly.

We refer to the set of attributes and properties as *system characteristics*:  $\Xi = \Theta \cup \Pi$ .

### 5.1.2.2 Influence relationships

Relationships between two system characteristics are present if a change in one property potentially influences the other. In the case study, properties *Battery capacity* and *Current drawn* could be considered two properties with a relationship in between (Figure 5.8): a change in the *Battery capacity* will have an impact to the *Current drawn* and vice versa.

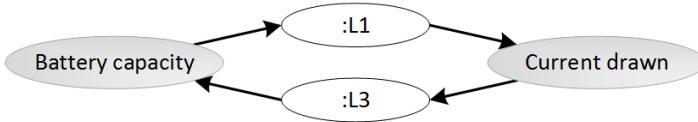


Figure 5.8: Influence relationship between two attributes.

A relationship  $r$  is formally defined as  $r \in R = (\Xi_i^j \subseteq \Xi, \Xi_m^n \subseteq \Xi, \lambda)$ , i.e.

- a set of influencer (input) properties  $\Xi_i^j = \{\xi_i \dots \xi_j\}$ ,
- a set of influencee (output) properties  $\Xi_m^n = \{\xi_m \dots \xi_n\}$ ,
- a level of precision  $\lambda \in \{\mathbb{L}1, \mathbb{L}2, \mathbb{L}3\}$ .

The three levels of precision have been defined in our work [42] and are as follows.

- L1: the **fact of influence** is known, its extent is not;
- L2: **sensitivity information** between two values is known;
- L3: the relationship can be expressed using an exact **mathematical relationship**.

Figure 5.8 shows a pair of attributes that mutually influence each other, albeit on different levels of precision. A change in the *Current drawn* has an L3 influence on the *Battery capacity* as follows:

$$\text{BatteryCapacity} \geq \int \text{CurrentDrawn}(t) dt.$$

The relationship in the other direction, however, cannot be determined in such details and thus, only constitutes an L1 relationship. In the running example, the relationship between the *Battery mass* and *Battery capacity* constitutes an L2 relationship: increasing the capacity requires increasing the battery mass, although the exact relation cannot be provided as batteries come in various architectures.

### Acausal influence relationships

Acausality provides compactness in terms of the notation of relationships: it enables modeling of N-ary relationships in a more convenient and readable fashion. Figure 5.9a shows an N-ary influence relationship, depicting a simple law of physics:

$$\text{BatteryMass} + \text{MotorMass} = \text{TotalMass}.$$

By assigning value to two of the three system characteristics, the third can be automatically calculated. The same information can be captured in an acausal way as presented in Figure 5.9b.

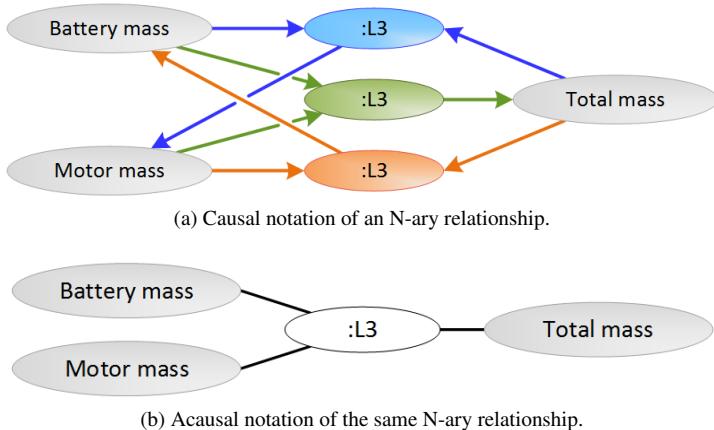


Figure 5.9: Causal and acausal notation of a relationship.

In our approach, we allow using acausal relationships, but translate them to the causal equivalent form when carrying out analyses. Formally, given an acausal influence relationship  $r$  of

level  $\lambda$  over a set of system characteristics  $\Xi'$ , the causality assignment maps  $r = (\emptyset, \Xi', \lambda)$  onto a set of relationships  $R' = \langle(\Xi'', \Xi''', \lambda)\rangle$ , such that  $\Xi' = \Xi'' \cup \Xi'''$ .

The causal equivalent can be unambiguously determined only in symmetric N-ary relationships, meaning any  $M$  number of the  $N$  system characteristics determine the remaining  $N - M$ , e.g. the one in Figure 5.9. In this case the causal equivalent will consist of  $\binom{N}{M}$  causal relationships.

### Change scope of system characteristics

By a change scope of a system characteristic  $\xi$  we mean a subset of properties  $\chi_\xi \subseteq \Xi$  potentially affected by a change in  $\xi$ . Given two system characteristics  $\xi_1$  and  $\xi_2$  directly linked by a relationship  $r(\Xi_i^j, \Xi_m^n, \lambda)$ , the change set is defined as follows.

$$\xi_2 \in \chi(\pi_1) \Leftrightarrow (\xi_1 \in \Xi_i^j \wedge \xi_2 \in \Xi_m^n)$$

That is, system characteristic  $\xi_2$  is in the change scope of system characteristic  $\xi_1$  iff  $\xi_1$  is an input system characteristic and  $\xi_2$  is an output system characteristic of a relationship. In the running example, the *Battery mass* property is in the change scope of *Battery capacity*.

The change scope is a reflexive and transitive relation, i.e.

$$\forall \xi \in \Xi : \xi \in \chi(\xi),$$

$$\forall \xi_1, \xi_2, \xi_3 \in \Xi : \xi_2 \in \chi(\xi_1) \wedge \xi_3 \in \chi(\xi_2) \Rightarrow \xi_3 \in \chi(\xi_1),$$

respectively. We use the following notation for the transitive closure of the change scope:  $\xi_3 \in \chi^+(\xi_1)$ .

#### 5.1.2.3 Intents

Intents capture the *motivation* of an activity with respect to a system characteristic or a relationship, such as *reading* and *modifying* an attribute. An intent  $i \in I$  is defined by the tuple  $(a, s, t_I)$ , where

- $a \in A$  is an activity;
- $s \in \Xi \cup R$  is the subject of intent, i.e. a system characteristic or a relationship;
- $t_I \in T_I$  is the type of intent.

We define four elementary intents for our approach:  $T_I = \{read, modify, check, contract\}$ , the first two being the typically occurring intents in standard engineering activities, while the latter two are specific to activities related to inconsistency management.

As discussed in Section 5, the main rationale behind explicitly modeling intents is that they carry valuable information regarding inconsistencies in processes, which enables reasoning about the origin and the potential management of inconsistencies. The inconsistency in the running example is possible to detect because of the exactly modeled pair of the read-modify intents on system characteristics that influence each other and activities that are

control-dependent. In Section 5.2 we formally characterize inconsistencies in terms of processes, system characteristic and intents.

#### 5.1.2.4 Typing of the property model

Intents relate properties to processes. In order to handle property models in a type-safe manner, however, attributes, properties and relationships have to be related to the type system defined by the language model as well.

We handle elements of the property model as special *process artifacts* that activities interact with. That is, following the definition of processes in Section 2.2:

$$\begin{aligned} \forall p \in P \forall s \in \exists \bigcup R : s \in D(p), \\ \forall p \in P \forall i \in I : i \in \Delta_d(p). \end{aligned}$$

Consequently, attributes, properties and relationships are typed by appropriate languages, e.g. *OWL* languages [15], for modeling properties, or *graphs*, *algebra* and *Forrester system dynamics* [72] for modeling relationships.

Intents are typed by an *intent language*  $T_I$ , i.e.

$$\forall T_I = \{i_1..i_n\} : T_I \in \mathcal{F}.$$

This means the language of intents in our approach can be aligned with the application domain of the problem at hand. The intents used throughout this paper are rather general and capture only access-change information over system properties.

#### 5.1.2.5 Modeling the running example

The metamodel of the property model is too complex to show an excerpt of it. The full, detailed metamodel, however, is shown in Figure A.3 of Appendix A.

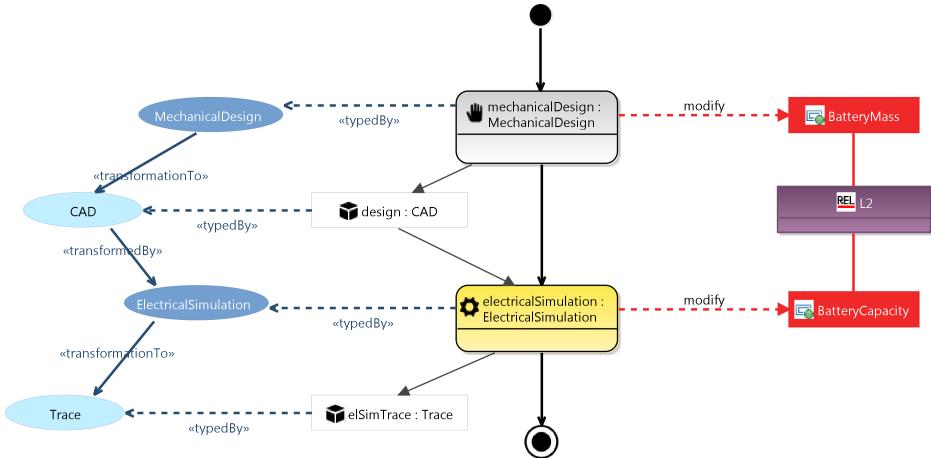


Figure 5.10: The running example in Figure 5.7, extended with attributes, constraints and intents.

Figure 5.10 shows the extended model of the running example. The *mechanicalDesign* activity has a *modification* intent on the *BatteryMass* attribute, because it is indeed the intention of the activity to set a new value to of the battery mass. Similarly, the *lectricalSimulation* activity has a modification intent on the *BatteryCapacity* attribute. Between the two attributes, an  $\mathbb{L}2$  relationship is captured, suggesting there is a sensitivity relationship between the two attributes, but it is not obvious to capture on the  $\mathbb{L}3$ .

### 5.1.3 Resources

The availability of resources impose constraints as of how well-performing a process can be. This is due to the fact that the level of parallelism of the process depends on the available resources. Some activities can be executed at the same time or at least in an overlapped fashion, but some must be executed in a sequential order.

When talking about resources, we mean both physical (e.g. a machine, a conveyor belt, etc) and logical resources (e.g. software licences). At this point, we only focus on *renewable* resources, i.e. the ones that become available for an activity again, once a preceding activity has been terminated.

When optimizing a process, we factor in the availability of resources and the resource demand of the process. To this end, we support modeling the resource demands and availabilities in our formalism.

To consider a process valid, all the resource *demands* have to be satisfied by the *available* resources; i.e. a valid *allocation* of the resources is required.

**Resource.** The resources are formalized as a set  $\Omega = \{\omega_1, \omega_2, \dots\}$ .

**Availabilities.** The availability  $b$  of resource defines how many units are available for the certain resource.  $b(\omega) : \omega \in \Omega \mapsto \mathbb{N}$ . As defined by [9], if  $b(\omega) = 1$ , then  $\omega$  is called a unary or disjunctive resource. Otherwise, as a resource may process several activities at a time, it is called a cumulative resource. The availability of all the resources is captured as  $B = \{b(\omega_1), b(\omega_2), \dots\}$ , i.e. a vector of natural numbers  $B(\Omega) : \Omega \mapsto \mathbb{N}^{|\Omega|}$ .

**Resource demands.** The resource demand  $K$  of an activity  $a \in A$  defines how many of the various types of resources an activity requires, and is represented as a vector of natural numbers over the resources:  $K(a) : a \in A \mapsto \mathbb{N}^{|\Omega|}$ , where the  $i$ -th element  $k_i$  is defined as  $k_i(a, \omega) : a \in A, \omega \in \Omega \mapsto \mathbb{N}$ .

**Allocation.** The resource allocation  $L$  of an activity  $a \in A$  and resource  $\omega \in \Omega$  defines how many units of  $\omega$  is  $a$  using.  $L(a) : a \in A \mapsto \mathbb{N}^{|\Omega|}$ , where the  $i$ -th element  $l_i$  is defined as  $l_i(a, \omega) : a \in A, \omega \in \Omega \mapsto \mathbb{N}$ .

**Valid allocation.** An allocation is valid iff the resource demands are met, i.e.  $\forall i = \{1..|\Omega|\} \in \mathbb{N} : l_i \geq k_i$ . Ideally,  $l_i = k_i$  is expected.

### 5.1.3.1 Modeling support

Figure 5.11 shows the relevant excerpt of the resource modeling part of the whole formalism. The full, detailed metamodel is shown in Figure A.3 of Appendix A.

As proposed, the *Resource* is the central concept of this facade of the modeling formalism. *Resources* provide an integer field to capture the available units of that resource. The allocation between activities and resources is accomplished using the *Allocation* type, linking an *Activity* with a *Resource*, while also explicitly stating the amount of allocated units.

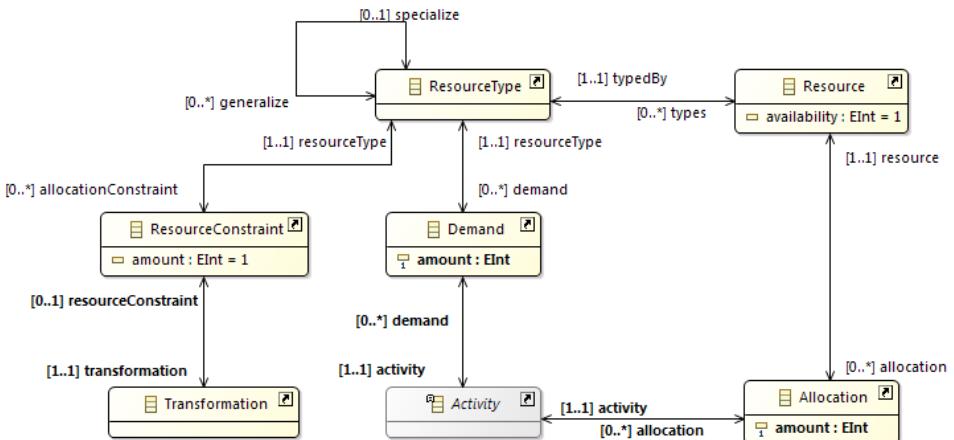


Figure 5.11: Excerpt of the resource modeling part of the formalism.

The demand is captured one meta-level above, as demands are stated against a specific type of resources. Hence the need for the explicit *ResourceType*. (Apart from the obvious need for a sound type system.) The *Demand* type then specifies the amount of units of a *ResourceType* required for an *Activity*. Through the typing relations in the FTG+PM, these relationships commute well, and the back-and-forth between meta-levels does not introduce any overhead. It is hidden from the user.

Finally, the formalism supports defining *ResourceConstraints* an additional meta-level higher: on the level of *Transformations*. In certain cases, it is known what resources a *Transformation* requires. This additional constraint must be met as well in order to achieve a valid process.

### 5.1.3.2 Modeling the running example

In the running example, the two activities can impose the following resource demands.

**mechanicalDesign:** requires a mechanical engineer, and a licence for a proprietary CAD tool.

**electricalSimulation:** requires an electrical engineer and a mechanical engineer, and a licence for the AMESim tool.

**Modeling the resources and their availabilities** There are four types of resources in this example: the *Mechanical Engineer*, the *Electrical Engineer*, the *CAD licence*, and the *AMESim licence*. Figure 5.12 shows how the modeling of the resources are accomplished using the modeling tool provided with the formalism. For demonstrative reasons, the example is modeled so that the demands can be satisfied.

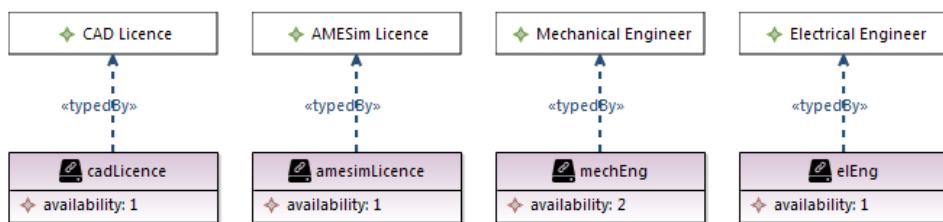


Figure 5.12: Resource types and resources of the running example.

**Modeling the demands** Now, that the types of resources are modeled, we can formalize the resource demands of the specific activities. Figure 5.13 shows the table-based syntax provided by the tool to model the unit-based demands for the various activities.

	CAD Licence	AMESim Licence	Mechanical Engineer	Electrical Engineer
mechanicalDesign	1		1	
electricalSimulation		1	1	1

Figure 5.13: Resource demands of the running example.

**Modeling the allocations** Finally, the allocations can be modeled or automatically inferred. A valid allocation is shown in Figure 5.14.

	cadLicence	mechEng	elEng	amesimLicence
mechanicalDesign	1	1		
electricalSimulation		1	1	1

Figure 5.14: Resource allocations of the running example.

#### 5.1.4 Cost models

After having modeled the process along with its type system, the relevant properties of the system engineered in the process, and the resources available for the process' activities, the last facility required to reason about various alternatives of the process, is modeling and simulating the performance of the process.

We are looking for an optimal process, and this is typically meant in a Pareto way, i.e. multiple performance factors result in a Pareto-front, from which any alternative is equally optimal.

We take a cost-based approach to the performance analysis of the process. This means, we allow modeling of various cost metrics, which are mapped to a common metric space

with an ordering operator. In practical terms, three *financial* aspect of the process can be modeled.

**Cost per execution time ( $c_e$ )** : the cost depends on the execution time of the process. A typical example is the salaries of the engineers required for the activity.

**Cost per unit ( $c_u$ )** : the cost is uniform for the activity, regardless of its execution. A typical example is the cost of materials required for the activity.

**Cost for presence ( $c_p$ )** : the cost is counted only once (or N times, in general scenarios) for the process, regardless the activity resource demands. A typical example is the cost of licences for engineering tools payed upfront.

The cost of the process, then is calculated as  $C(p) = c_e(A) + c_u(A) + c_p$ . Additionally,  $C(p) : p \mapsto \mathbb{R}^+$ .

Costs serve as a basis for quantifying the differences between various process alternatives. In our current work, we approximate costs by the transition time required for single activities. Non-linear processes, i.e. the ones with directed loops, are typical in engineering scenarios. In these cases, the cost of a process is a non-deterministic value that can be obtained by appropriate simulations.

#### 5.1.4.1 Modeling support

Figure 5.15 shows the relevant excerpt of the cost modeling part of the whole formalism. The full, detailed metamodel is shown in Figure A.5 of Appendix A.

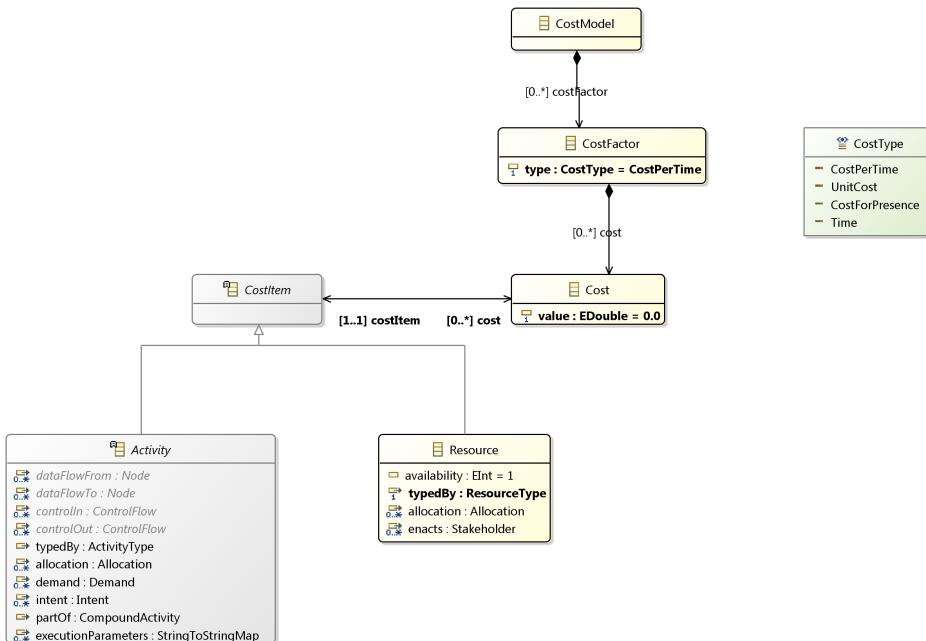


Figure 5.15: Excerpt of the cost modeling part of the formalism.

The formalism enables specifying various *CostFactors* of different *CostTypes* for the *CostItems*, i.e. *Activities* and *Resources*.

### 5.1.4.2 Modeling the running example

Modeling the costs is accomplished using a table-based syntax provided by the tool, as shown in Figure 5.16. Activities *mechanicalDesign* and *electricalSimulation* are associated with *Time* types of costs. This means, the former one takes 40 hours to finish, while the latter one takes 1 hour to finish. Additionally, the *mechEng* and *elEng* resources (i.e. the Mechanical Engineer and the Electrical Engineer, respectively), are associated with a time-based wage. Both make 30 EUR per hour in this example. (Which is a realistic gross amount from the company's perspective.) Finally, the licence fees for executing the activities are captured as one-time costs for the whole process. The *cadLicence* costs 2000 EUR, while the *amesimLicence* costs 4000 EUR.

	$\diamond$ mechanicalDesign	$\diamond$ electricalSimulation	$\diamond$ cadLicence	$\diamond$ mechEng	$\diamond$ elEng	$\diamond$ amesimLicence
$\diamond$ licenseCost : CostForPresence			2000.0			4000.0
$\diamond$ wage : CostPerTime				30.0	30.0	
$\diamond$ time : Time	40.0	1.0				

Figure 5.16: Associated costs of the running example.

It is thanks to resource allocation model that these cost factors eventually can be composed into a single composite cost metric, i.e. the monetary costs in EUR, in this case. The eventual cost of this simple example will be calculated as follows.

- The cost of the mechanical design activity:  $40 \times 30$  in wages and  $1 \times 2000$  in licence fees.
- The cost of the electrical design activity:  $2 \times 1 \times 30$  in wages and  $1 \times 4000$  in licence fees.

Resulting in 7260 EUR in sum monetary costs, and 41 hours of time costs.

In the later sections, we will show two simulation methods for costs. A quick and simple method is shown from the earlier stage of this research in Section 5.2.3.5; an advanced, state-of-the-art simulation method is shown in Section 5.5.

### 5.1.5 ISO/IEC/IEEE 42010:2011 compliant viewpoints

It falls outside of the scope of this thesis, but our formalism also provides modeling support for ISO/IEC/IEEE 42010:2011 compliant view/viewpoint specifications. [102]

As outlined in Section 2.1.5 and additionally discussed in Section 4.5, MPM and multi-view settings are strongly related. Our intention with the added viewpoint modeling facility is to enable the explicit modeling of the information viewpoints entail, including stakeholder concerns, and formalism/tool selection for certain types of engineering activities. These information do not fit the FTG+PM concept in its classic form, and instead of forcing

these information into the FTG+PM, we provide a link to the architecture imposed by the ISO/IEC/IEEE 42010:2011 standard.

In Figure A.6 of Appendix A we present the metamodel for viewpoint modeling scenarios. Given that this topic has been never researched thoroughly, we do not elaborate on it further and leave it as a future work.

## 5.2 Off-line inconsistency management

In this section, we present **Contribution 4** of this thesis: a formalism for modeling complex engineering processes. The majority of this section has been published in [41].

The management of inconsistencies is achieved by selecting the appropriate techniques from a catalogue of management patterns and applying them on the original unmanaged process to achieve an optimized and managed one. Typical patterns of inconsistency management include re-ordering activities of a process, ensuring property checks around inconsistency-prone regions and using design contracts [159].

We approach the problem of managing inconsistencies as a process optimization problem. There are multiple alternative inconsistency management patterns that can be applied for the same type of inconsistency. Selecting the appropriate one happens by considering a performance metric of choice the process is the most optimal according to. Applying the elementary inconsistency management patterns and process optimization patterns, leads to different process alternatives. These alternatives span the *process space*, that has to be efficiently traversed in order to find the best appropriate solution.

We formalize this problem as a constraint solving and optimization problem as follows.

$$\begin{aligned} & \underset{p}{\text{minimize}} && \text{cost}(p \in P) \\ & \text{subject to} && \{IC\} \equiv \emptyset, \\ & && v(p) = 1. \end{aligned}$$

where  $v(p) : p \in P \mapsto \mathbb{B}$  is the indicator function of the validity (i.e. well-formedness) of a process  $p$ . We also demand a process in which every potential inconsistency is managed, i.e. the set of the unmanaged inconsistencies is empty:  $\{IC\} \equiv \emptyset$ .

### 5.2.1 Patterns of inconsistency

In the following, we identify cases when inconsistencies may occur. We formalize this information in terms of pairs of activities, the related properties and intents. Generally, inconsistencies are introduced when an activity *modifies* a property that is accessed by another activity. A more formal definition can be given by distinguishing between activities situated in a sequential order and in parallel branches of a process. By sequential and parallel activities we mean the following.

**Sequential** :  $\forall a_1, a_2 \in A : seq(a_1, a_2) \Leftrightarrow a_2 \in \Delta_c^+(a_1)$ ;

**Parallel** :  $\forall a_1, a_2 \in A : \text{par}(a_1, a_2) \Leftrightarrow a_2 \notin \Delta_c^+(a_1) \wedge a_1 \notin \Delta_c^+(a_2)$ .

That is, two activities are said to be sequential iff one activity is transitively reachable from the other via the control flow of the process. If no such relation exists (in any direction), the activities are said to be parallel.

### 5.2.1.1 Sequential case

Given a pair of activities  $a_1, a_2 \in A : \text{seq}(a_1, a_2)$ , property  $\pi$  is said to be exposed to a potential inconsistency due to insufficient inconsistency management in the following case.

$$\exists(\pi' \in \Pi, i_1, i_2 \in I) : i_1(a_1, \pi, t_1) \wedge i_2(a_2, \pi', t_2) \wedge \pi \in \chi^+(\pi') \wedge t_1 = \text{read} \wedge t_2 = \text{modify} \Rightarrow \exists ic(\pi) \in IC,$$

where  $IC$  denotes the set of unmanaged inconsistencies.

That is, property  $\pi$  is exposed to a potential inconsistency if activity  $a_1$  first accesses it with a *read* intent and subsequently activity  $a_2$  *modifies* property  $\pi'$ , while property  $\pi$  is in the change scope of  $\pi'$ . The relation does not hold the other way around, i.e. by first modifying and subsequently reading a property does not lead to inconsistencies.

As a consequence of the reflexivity of the change scope, the above definition applies on cases where the same property is being read and modified as well.

### 5.2.1.2 Parallel case

Given a pair of activities  $a_1, a_2 \in A : \text{par}(a_1, a_2)$ , property  $\pi$  is said to be exposed to a potential inconsistency in the following case.

$$\exists(\pi' \in \Pi, i_1, i_2 \in I) : i_1(a_1, \pi, t_1) \wedge i_2(a_2, \pi', t_2) \wedge \pi \in \chi^+(\pi') \wedge t_2 = \text{modify} \Rightarrow \exists ic(\pi) \in IC.$$

The definition is different from the one in the sequential case in not being specific about the type of intent  $i_1$ . This is because of the inconclusive ordering of  $a_1$  and  $a_2$  due to their parallel relation. Since the two activities may access the related properties in any order, the cases of potential inconsistencies cannot be narrowed to a specific ordering of read-modify intent pairs. That is, inconsistencies may arise if any of the two activities *reads* property  $\pi$ , while the other one *modifies*  $\pi'$ .

## 5.2.2 Patterns of inconsistency management

We use four typical inconsistency management patterns in our approach. This catalogue of patterns is, however, extensible in the prototype tooling.

### 5.2.2.1 Reordering and sequencing

Reordering and sequencing aim to modify the control flow in order to avoid inconsistencies.

Given a sequential case, i.e.  $a_1, a_2 \in A : seq(a_1, a_2) \Rightarrow \pi \in IC$ , the reordering strategy would swap  $a_1$  and  $a_2$ , i.e.  $seq(a_1, a_2) \rightarrow seq(a_2, a_1)$ , to utilize that the appropriate order of read-modify intents does not lead to inconsistencies, as shown in Section 5.2.1.1.

In parallel cases, i.e.  $a_1, a_2 \in A : par(a_1, a_2) \Rightarrow \pi \in IC$ , the sequencing strategy would try every possible order of the activities and eventually select the one that leads to the most optimal process, i.e.  $par(a_1, a_2) \rightarrow seq(a_1, a_2) \vee seq(a_2, a_1)$ .

Reordering and sequencing are easy-to-apply and inexpensive patterns as they do not require introducing additional management activities. Both patterns work well in simple cases; in more complex processes, however, both patterns tend to introduce other inconsistencies.

### 5.2.2.2 Property check

Property checking is used to ensure no inconsistencies are introduced on specific sections of the process. A special activity  $a_{check}$  is added to the process that accesses the unmanaged properties with a *check* intent. If the result of the check is satisfactory, the process continues with the subsequent activities; in the case of a failed check, however, the process would fall back to the latest point where the inconsistency is not yet present and facilitate a re-iteration loop.

The property check pattern is a typically expensive management pattern as it introduces directed loops in the design processes and therefore, makes processes inherently non-deterministic.

### 5.2.2.3 Contracts

In a contract-based approach [203], the stakeholders would agree on acceptance criteria of specific properties *before* executing specific design activities. A special activity  $a_{contract}$  is added to the process to represent the contract negotiation phase. The activity accesses the unmanaged properties with a *contract* intent. The contract is respected during the activities, thus providing means to avoid inconsistencies.

### 5.2.2.4 Assumptions

A less rigorous approach to contracts is also possible by making an educated guess about the shared properties. In the parallel case:  $a_1, a_2 \in A : par(a_1, a_2) \Rightarrow \pi \in IC$ , one of the parallel activities makes assumptions about the properties that will be modified by the other activity. However, these assumptions need to be checked once the process rejoins both branches. The benefit of the pattern is that only one of the branches has to be re-executed if the assumption proves to be invalid, i.e. an inconsistency may occur.

### 5.2.3 Solving the process optimization problem by multi-objective process-space exploration

To solve the process optimization problem, we employ a *search-based* technique. The set of every possible processes is considered and an algorithm searches through this set to find the possible process alternative(s). Figure 5.17 shows this concept. The search starts from the original process  $p$ . In each step, any process transformation rule  $r \in R$  can be applied onto the current process alternative. The set of transformation rules  $R$  consists of two types of transformations: inconsistency management transformations and process performance enhancing transformations. Eventually, the search algorithm finds the (at least locally) optimal process alternative  $p^*$ .

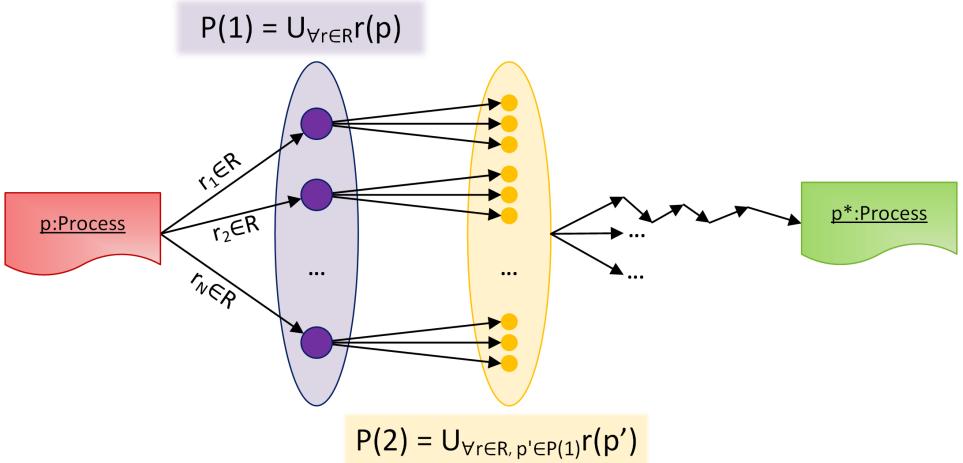


Figure 5.17: Detailed overview of the search process.

In step  $M$ , the set of process candidates is  $P(M) = \bigcup_{r \in R, p' \in P(M-1)} r(p')$ . Assuming that the number of applicable transformation rules is  $|R| = N$ , in the  $M$ th step, the set of process candidates is  $N^M$ .

**Process space.** By process space  $\Sigma_P$ , we mean the set of process candidates that are reachable from the initial process  $p$  via a sequence of transformation rules  $r_1, \dots, r_M$ . Formally,  $\Sigma_P = \bigcup_{i=1}^M P(M) = \bigcup_{i=1}^M \bigcup_{r \in R, p' \in P(M-1)} r(p')$ .

The process space is typically an infinite set and grows exponentially in each iteration. Therefore, it is infeasible to carry out the search in an exhaustive fashion. *Design-space exploration* techniques, however, help tackling this issue. We employ a rule-based multi-objective design space exploration (DSE) approach to search through the process space.

**Design space exploration.** Design space exploration aims at searching through various models representing different design candidates to support activities like configuration

design of critical systems or automated maintenance of IT systems [87], or to obtain deeper insight into the design problem and better formulate the optimization problem [209]. In model-driven settings DSE is typically achieved by gradually applying model transformation rules to the source model to find instance models that are reachable from the source model, while satisfying a set of constraints. In general, design space exploration is done by extending the abstract model by implementation constraints to identify suitable solutions. [161] The traversal process is guided by hints or smart search heuristics to avoid unpromising subsets of the design space.

In our case, the previously defined *process space* is the design space spanned by the original process and the set of inconsistency management and process performance enhancing model transformations.

The multi-objective nature of the DSE refers to the fact that there are multiple optimality criteria (some mandatory, some optional), which imply a Pareto-front of equally optimal solutions.

Figure 5.18 shows the overview of the DSE approach used to search to the process space.

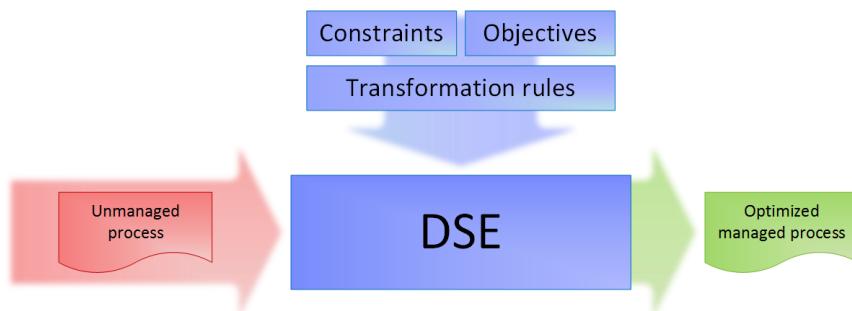


Figure 5.18: Structural overview of the DSE approach.

The exploration mechanism takes the original *unmanaged process* and the *property model* as an input and produces an *optimal managed process* as a series of *model transformations* applied on the original process. (The property model is left intact as it reflects domain knowledge and as such, typically should not be changed because of a single process.) The exploration process is guided by mandatory *constraints* and optional *objectives*.

For capturing the transformations, constraints and objectives, explicitly modeled *graph queries* are used. Constraints and objectives are evaluated by matching the graph queries against the process model at hand and counting the matches. Transformations employ the graph queries as the left-hand sides of the transformation rules.

We use the VIATRA DSE framework [2] for implementing this feature in our tool. Figure 5.19 shows the architecture of the DSE engine in the core of our prototype tooling. The inconsistency and management catalogues, as well as cost models are fully extensible, i.e. new inconsistency and management patterns can be formalized by the appropriate graph query and transformation languages, and costs can be evaluated using other approaches than the ones presented here.

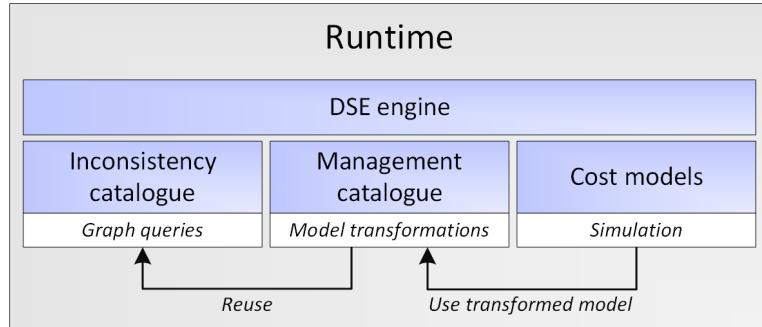


Figure 5.19: Architectural overview of the design space exploration component in the core of the prototype tooling.

### 5.2.3.1 Inconsistency catalogue

The patterns of inconsistencies are captured by graph queries over the input model. In our prototype tool, we use the VIATRA Query Language (formerly EMF-IncQuery) [184] for this purpose. Listing 5.1 presents the inconsistency pattern matched on the running example.

```

1 pattern unmanagedReadModify(
2     a1:Activity, p1:Property,
3     a2:Activity, p2:Property){
4     find readModifySharedProperty(a1, p1, a2, p2);
5     checks == count find checkProperty(a2, _, p2);
6     contracts == count find contract(_, a1, p1);
7     check (checks+contracts==0);
8 }
```

Listing 5.1: The read-modify inconsistency pattern.

The pattern reflects the left-hand side (LHS) of the general transformation rule in Section 5.2.3.4. In Line 7, properties  $p1 \in R^+(p2)$  and activities  $a2 \in \Delta_c(a1)$  accessing the two properties with the respective *read* and *modify* intents are identified. Subsequently, in Lines 8-9, the number of applied inconsistency management patterns is determined. In case there is no management pattern applied (Line 10), the pattern is matched and the match set requires an inconsistency management pattern to be applied.

The detailed inconsistency patterns are listed in Listing 5 of Appendix B.

### 5.2.3.2 Management catalogue

Management patterns are captured as model transformations over the model. In accordance with Section 5.2.3.4, the LHS of the transformation rules consist of the previously defined inconsistency patterns; while the right-hand side (RHS) defines how the specific inconsistency should be handled by using one of the management patterns described in Section 5.2.2. Allowed LHS-RHS combinations are specified in terms of VIATRA Model Transformations [18], that enable directly reusing the previously defined graph queries as the LHS.

### 5.2.3.3 Constraints and objectives

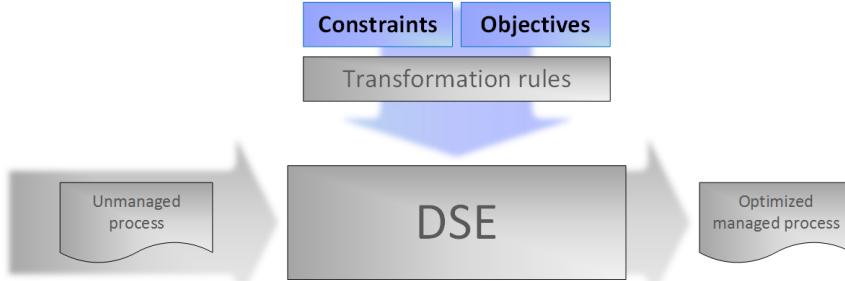


Figure 5.20: Constraints and objectives of the DSE.

Constraints and objectives are used to guide the exploration process and evaluate the solution candidates. We constrain the set of solutions to processes that are valid and have no unmanaged inconsistencies.

As the objective function, the cost of the process is used. Since the cost of non-linear processes (i.e. the ones featuring directed cyclic graphs) is not deterministic, simulations of various kinds can be used to obtain the cost, such as event queueing networks or discrete event simulations.

Listing 9 of Appendix B shows the constraints and objectives used in our framework in greater details. Here, we show one typical

### 5.2.3.4 Transformation rules

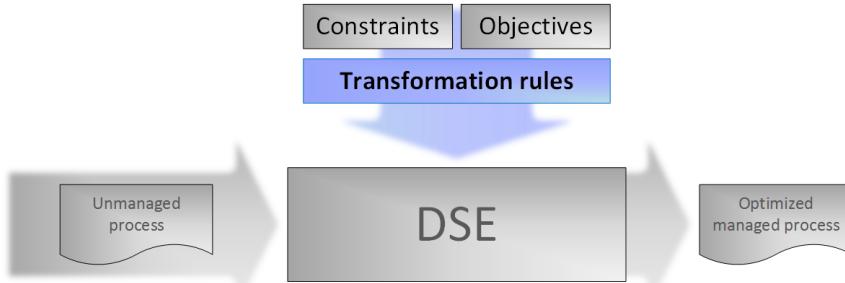


Figure 5.21: Transformation rules of the DSE.

The purpose of using model transformations is twofold. We use them to augment the process with inconsistency management techniques, but also for rewriting the process into a better performing process. An example for the latter one is parallelizing as many activities as possible. Of course, this will affect the applicable inconsistency management techniques,

and therefore, the execution and evaluation of these transformations must be achieved in a coupled way.

### Management transformation rules

Transformation rules aiming to augment the process with inconsistency management techniques, are derived from the inconsistency patterns (Section 5.2.1) and management patterns (Section 5.2.2). A transformation rule is defined as

$$\begin{aligned} \exists ic \in IC : ic(\omega \subseteq P) \wedge \\ \nexists m \in M : m(ic(\omega \subseteq P)) \\ \rightarrow apply(m'(ic(\omega \subseteq P))), m' \in M. \end{aligned}$$

That is, if an inconsistency pattern  $ic$  is detected on a subset  $\omega$  of the process model  $P$ , and there is no corresponding management pattern  $m$  detected for the same subset of elements, then an appropriate management pattern  $m'$  is applied to the inconsistency.

Listing 5.2 shows the abstract structure of the transformations used for the running example. The detailed management transformation rules are listed in Listing 7 and Listing 8 of Appendix B.

```

1 rule manageWithContract(ic: unmanagedReadModify) {
2   // apply contract on ic
3 }
4 rule manageWithCheck(ic: unmanagedReadModify) {
5   // apply propertyCheck on ic
6 }
7 rule manageWithReorder(ic: unmanagedReadModify) {
8   // reorder(ic)
9 }
```

Listing 5.2: Management alternatives of the read-modify inconsistency pattern.

### General transformation rules

The overall goal of our approach is to find better processes, with respect to a goal function, that create correct products. Apart from the transformations specific to inconsistency management, therefore, we also use transformations that manipulate the structure of processes, such as adding and removing control flow between activities, arranging activities into sequences or parallel branches, etc. There is no restriction on how far the exploration strategy can go in restructuring the process, as it is determined in the exploration phase by considering that every inconsistency has to be managed. By making certain activities parallel, more linguistic and semantic overlap exists at the same instant in the process and thus making the process more vulnerable to inconsistencies.

Note that certain applications of this pattern are not usable. For example, the parallelization of a design activity cannot be parallelized with the subsequent simulation of the created model.

### 5.2.3.5 Design space exploration mechanism

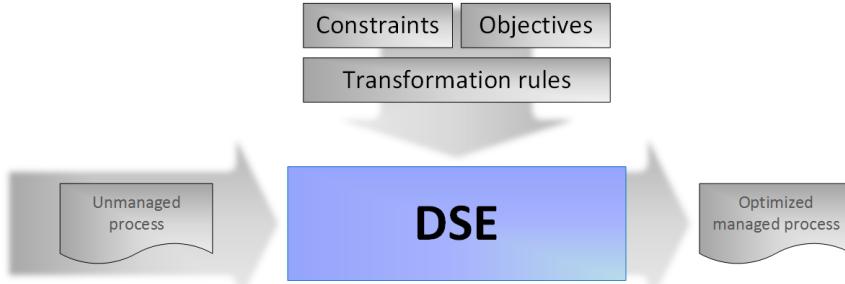


Figure 5.22: Design space exploration mechanism.

Figure 5.23 shows the process model of the DSE mechanism, corresponding to the overview in Figure 5.22. The DSE process starts with taking the *original process* as an input. First, it is determined if the mechanism should terminate due to a *timeout*. If not, the process proceeds with *selecting the model transformation rules  $R'$*  out of every transformation rule  $R$  that are applicable for the process model at hand. Subsequently, the process *evaluates the criteria* against the *current process candidate* at hand; the criteria being the meeting of the *Objectives* while respecting the *Constraints*. The result of the evaluation is a *criteria metric*. This metric is used do determine whether the current branch of the search tree should terminate, i.e. *cut off*. If not, the exploration proceeds with *applying one transformation rule  $r \in R'$* . This results in updating the *current process candidate*. Subsequently, the model undergoes a *validation* step, and if it is determined to be valid, it is also evaluated whether it is a *solution*. If it is, the exploration process terminates with the *current process candidate* being promoted to the *optimized process*.

The DSE mechanism can also be parametrized so that not only one solution but a set of alternative solutions is returned. We opted for the simpler exploration process in order to simplify the proof-of-concept implementation.

**Search strategies** Traversing the design space in an efficient manner is a key in DSE. Search strategies help identifying the viable directions to be navigated towards in the search space. (Or dually: pruning the uninspiring branches of the search tree, and that, in a relatively early stage of the traversal.)

In our tooling, we use a hill climbing strategy to guide the exploration process, which comes out of the box with the employed DSE framework. [87] The search strategy takes the simulated performance value of the process as a guiding metric.

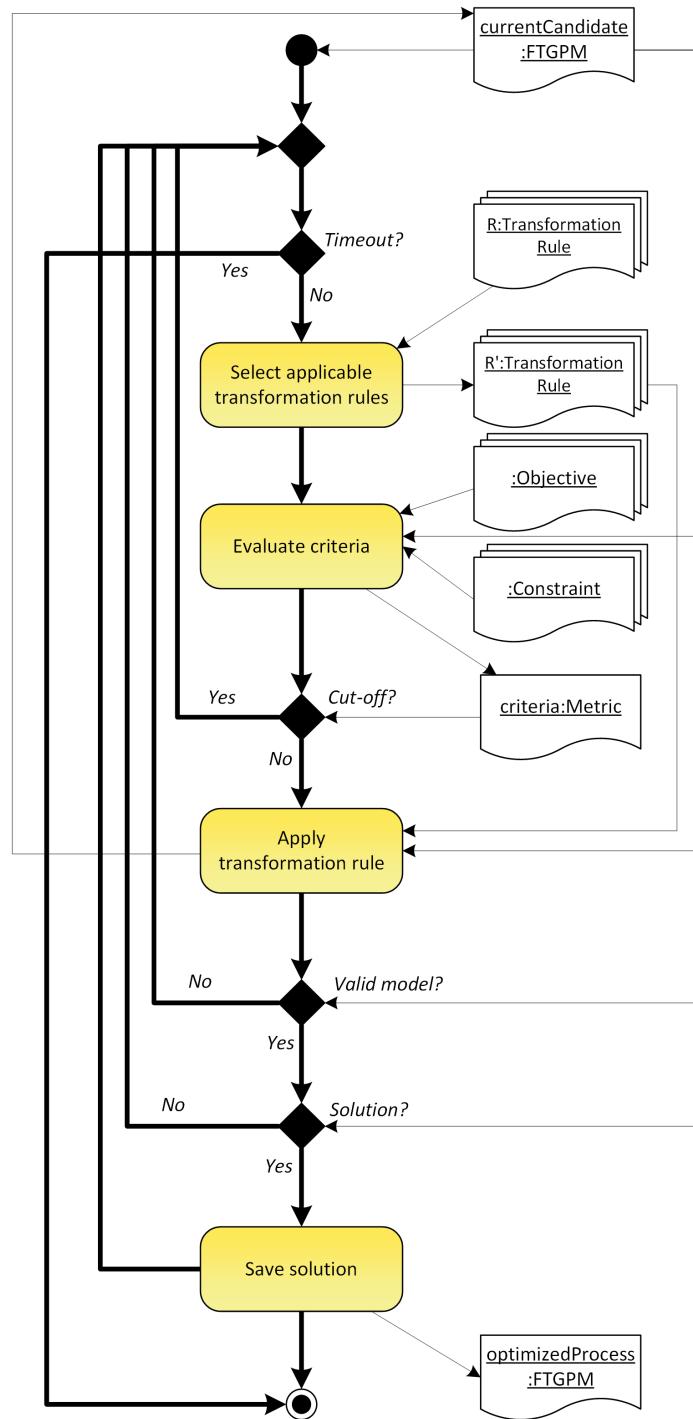


Figure 5.23: The PM of the DSE mechanism.

**Performance simulations of the process** Performance simulations of the process are required for guiding the search. At an earlier stage of this research, we provided two sub-optimal techniques for simulating performance. In this section, both are presented. At a later stage of this research, a much more efficient, automated simulation approach has been provided. That approach is discussed in greater details in Section 5.5.

**Fixed loop iterations** In *fixed iteration simulations*, the loops of the design process are identified and simulated with a fixed amount of iterations, shown in algorithm 1, resulting in a process cost as follows

$$\forall p \in P : c(p) = \sum_1^{i=|A(p)|} c(a_i)n(a_i).$$

Loops are detected by a graph pattern matcher. If a loop is detected between two activities, the costs of activities in the loop are weighted by the number of iterations  $N$  and added to the sum cost. The parameter is to be set by a domain expert. In our experiments, we used 3–5 iterations as the typical values.

---

**Algorithm 1** The fixed iteration simulation algorithm.

---

```

1: procedure SIMULATE( $N$  : int)                                 $\triangleright N$  is the number of iterations
2:    $cost := 0$ 
3:   for  $a_1, a_2 : A$  do
4:     if loop( $a_1, a_2$ ) then                                      $\triangleright a_1$  and  $a_2$  form a loop
5:       for  $a_3 : A \in \text{loop}(a_1, a_2)$  do                          $\triangleright a_3$  is in the loop
6:          $cost := cost + c(a_3) \times N$                                 $\triangleright$  add cost of  $a_3$  to cost
7:       end for
8:     end if
9:   end for
10:  return  $cost$ 
11: end procedure

```

---

**Event queueing network based simulation** In event queueing network (EQN) based *stochastic simulations* [82], the decision of re-iterating over a loop is simulated with sampling from a probabilistic distribution. We carried out our early experiments using the SimEvents toolbox [126]. While stochastic simulations offer more precise results in terms of simulating the costs, they are also more demanding in terms of computation power and time.

Here, we provide mapping rules for translating the engineering process onto the SimEvents formalism.

- **Activities** are translated to a Server processing a single token in the SimEvents formalism. The service time of the token in the server is based on the provided cost. The service time is either a constant value or a value sampled from a distribution.
- **Fork** nodes are translated to replicate nodes that process an incoming token (and all of its properties, like the total service time) to each of the outgoing branches.
- The **Join** node uses a combination of queues to let the tokens wait for the other branches. An entity combiner combines all tokens when they are available.

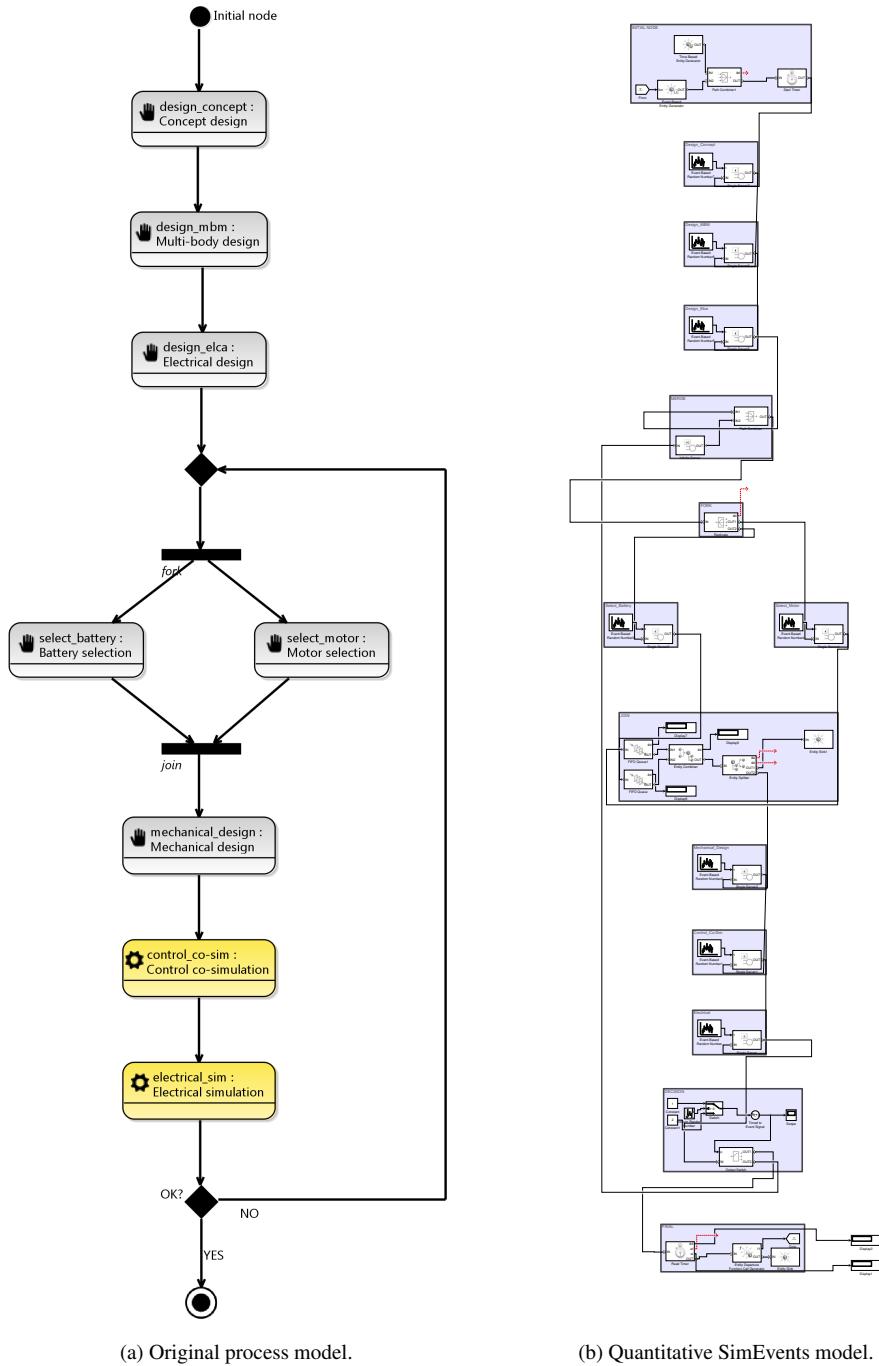


Figure 5.24: A process model and a quantitative SimEvents simulation model, depicting the parallel motor and battery selection steps.

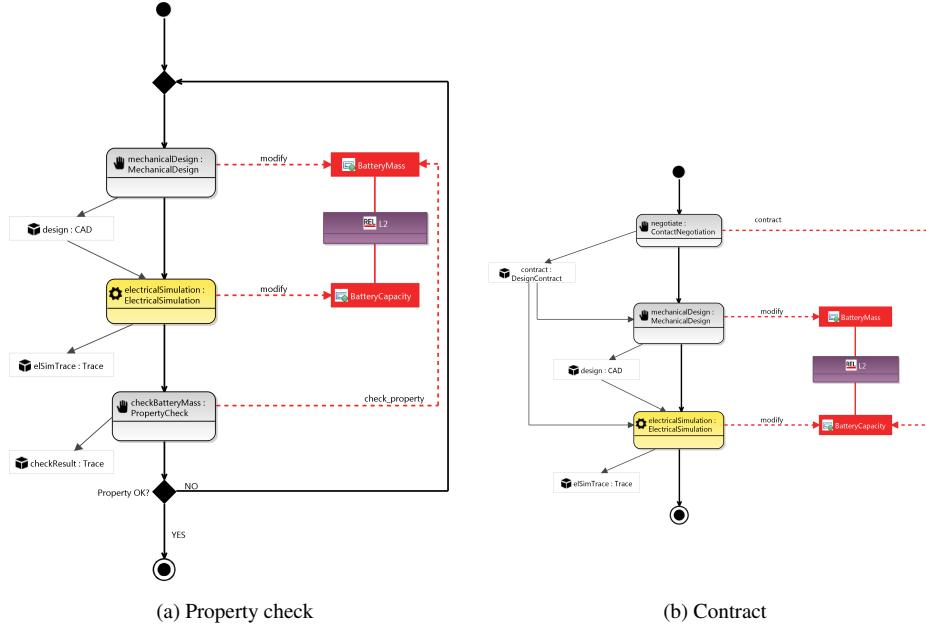


Figure 5.25: Two managed alternatives of the process in Figure 5.10.

- **Decision nodes** are translated to an output switch element that routes to the available paths. The chosen path is sampled from the information provided in the process model. The merge node uses a path combiner to combine the incoming paths.

Because we only allow for a *single process* to be executed at the same time, the logic in the final node allows for the creation of a new token in the initial node. The *control flow* between these newly created entities is equal to the control flow edges in the process model. For each of the tokens, the total service time is recorded. As SimEvents is a stochastic formalism, multiple tokens are used to calculate the total cost of the process. We use the average service time of all tokens as the cost of the process.

**Results of the exploration** After exploring the space of alternative solutions and ranking them based on the performance of the process, the managed process is obtained. Figure 5.25 presents two managed alternatives to the running example in Figure 5.10.

Figure 5.25a shows as an allow-and-resolve type of inconsistency technique, property checking is executed after the location of the potential inconsistency. The manual *check-BatteryMass* activity accesses the potentially inconsistent property *Battery mass* with a *check\_property* intent. The *checkBatteryMass* activity is typed by a special transformation: *PropertyCheck*. This transformation expects a set of properties on the input, a model to check the properties in, and a simulation technique; and produces a correspondent performance metric, a trace of the simulation, from which the satisfaction of the property, or the validity of the attribute can be inferred. Subsequently, a decision node is added to the control flow to enable a backward loop in case the check fails (*NO*) and to proceed if the check succeeds (*OK*).

Applying contracts (Figure 5.25b) is an avoidance style inconsistency technique and therefore, it manages inconsistencies before they can manifest. The *negotiate* activity captures the step when a handful of engineers *negotiate for a contract* [203]. The activity outputs an artifact of type *DesignContract*, containing assumptions and guarantees w.r.t. some properties of the system. In this case, the *BatteryCapacity* is the attribute to formulate assumptions and guarantees about. This means, the *mechanicalEngineering* step will **not** introduce inconsistencies because of the battery's capacity when selecting the battery.

Depending on their costs, any of these solutions can be applicable to the specific problem.

## 5.3 Process enactment

In this section, we present **Contribution 5** of this thesis: a formalism for modeling complex engineering processes. The majority of this section has been published in [43].

Process enactment is commonly defined as the use of software to support the execution of operational processes, which enables mixing automated and manual activities. [35] Our framework provides an engine for enacting previously defined (and optimized) processes with the additional support for interoperability with a selection of engineering tools, such as Matlab/Simulink or Papyrus. During the enactment, the constraints of the system's parameters are continuously monitored. By employing symbolic mathematics, constraints can additionally be maintained in an incremental fashion and used for guiding the engineering work. Whenever the value of a system attribute can be derived from a combination of constraints and previously assigned attributes, the engine will provide this information to the stakeholder, thus aiding engineering decisions.

In this section, we briefly present how the enactment of the previously modeled process (Section 5.1) is carried out while enforcing a consistent state across the models. Our custom process enactment engine with fully modeled execution semantics is presented, as well the algorithm used for detecting inconsistencies during the enactment of the process. To facilitate the interplay in real engineering settings, we provide integration with multiple tools and frameworks, which will be discussed briefly as well.

### 5.3.1 Architecture

Figure 5.26 shows the architecture of the process enactment engine. The engine is initialized by the *Process model*, defined previously in Section 5.1. An explicit *Enactment model* augments the *Process model* with the notion of tokens and activity states (Figure 5.27), to be able to define the execution semantics. Execution semantics are defined by explicitly modeled *Transformation rules*.

The architecture has been implemented on top of the Eclipse platform. The Eclipse Modeling Framework (EMF) [56] is used for modeling purposes, while the model transformations have been realized using the VIATRA framework [18].

Activities of the process, especially the automated ones, often execute simulations and calculations over models on external storages by using external tools. For that, interop-

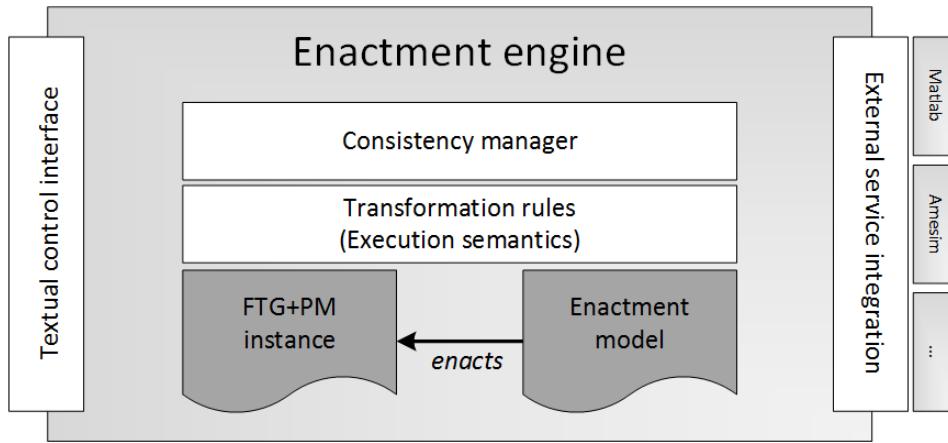


Figure 5.26: Architectural overview of the enactment engine.

erability with a representative set of services is provided (*External service integration*). Our framework currently provides scripting support for Matlab/Simulink, and Amesim of Siemens/LMS through its native API. Executable pieces of Java code or Python scripts are supported and executed during the appropriate phases of the enactment.

A vital contribution of the stack is the *Consistency manager*, which features a symbolic solver for detecting inconsistencies. For this purpose, the SymPy [176] framework for symbolic mathematics is used. In Section 5.4.3.1, the algorithm of the solver is discussed in greater detail.

### 5.3.2 Execution semantics

The execution semantics of the FTG+PM have been discussed previously in [50]. Here, we give a brief overview and focus on the main specificities in our current framework.

Since the core of enactment engine is fully modeled, the execution semantics are given by reactive live model transformations. Figure 5.27 shows the metamodel of the enactment engine. A *ProcessModel* (i.e. a full FTG+PM) is given to the compiler which creates an instance of the elements shown in green. During the enactment, a set of *Tokens* define the marking of the process, i.e. the active *Nodes* at a given moment.

A *Token* also equips *Activities* of the process with additional semantics regarding the state of their execution, modeled by *ActivityState*. This is required because the execution of *Activities* is not instantaneous.

- When a *Token* is moved to a new *Activity*, the *Activity* becomes *Ready*. The stakeholders and tools required to execute the *Activity* can be notified, the required models can be loaded into the tools.
- When the actual work in the *Activity* begins, the *Activity* becomes *Running*. This state can last for longer periods, especially in resource-intensive simulations or manual modeling activities, which may take days or weeks.

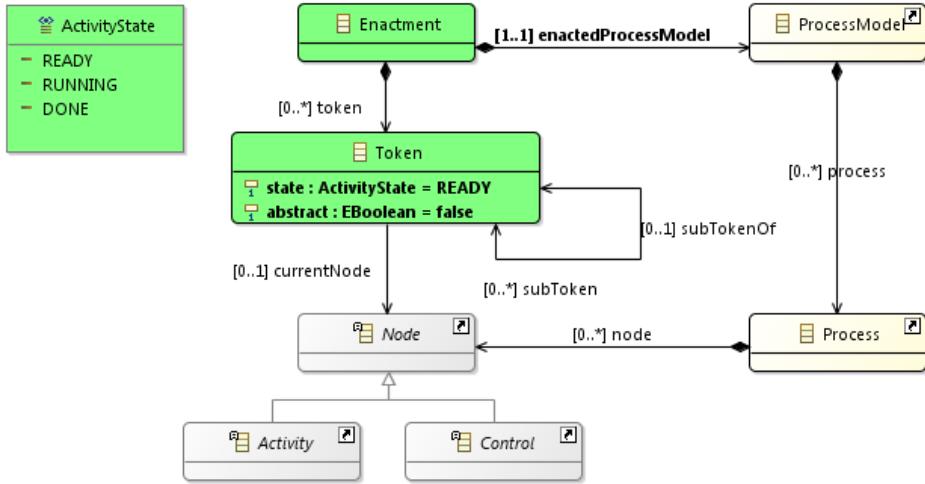


Figure 5.27: Metamodel for the enactment (green) along with the characteristic parts of the process metamodel.

- When the actual work in the *Activity* is finished, the *Activity* becomes *Done* and the process can move on.

### 5.3.3 Transformation rules

**Definition 1** (Marking of the process). *By marking  $M$  of the process we mean the function  $M : N \rightarrow \mathbb{Z}$ , where  $N$  denotes the set of the Nodes of the process with an integer number  $\mathbb{Z}$  of Tokens in it. A process is considered to be unmarked if there are no tokens present in it.*

**Initialization** is a transformation which takes an unmarked process and transforms it into a process with an initial marking, i.e. with one token in its initial node.

**Finishing** is a transformation which takes a process with a final marking (i.e. every token in the final node) and transforms it into an unmarked process.

**Fork** is a transformation which takes exactly one token and produces a token for each parallel branch starting from that fork node. The input token is marked *abstract* (see Figure 5.27) and kept (hidden) in the model, while the newly created tokens are defined as *subtokens* of the input token, so that they can be identified once they have to be joined at the end of the parallel branches.

**Join** is a transformation which takes a token from each of its incoming parallel branches and joins those tokens. In a valid process model, the tokens to be joined must be the subtokens of the same (now abstract) parent token. The join is achieved by locating the parent token, placing it into the join node, marking it as not abstract, and removing the subtokens.

**Step** is a transformation which moves a token from a node to a consecutive node, while respecting the previous rules of forking and joining.

### 5.3.4 Implementation

The transformation rules are captured using the VIATRA Transformation Language. We only show one of them, the rest is shown in Listing 10, Listing 11, Listing 12 and Listing 13 of Appendix B.

Listing 1 shows the Fork transformation rule. The rule defines its *name* as “forkable”; its *precondition* being a match of the model query called *forkable*; and the *action* to be executed upon a matching precondition. The action first logs the information of being executed to the console. (Line 3.) The token in the given node is being marked as *abstract*. (Line 5.) Subsequently, a loop goes through all the outgoing control flows from the given node (Line 7), and a new non-abstract token is being created for each subsequent node (Lines 8-11). If the subsequent node is an *Activity* (as opposed to being a control node), the activity is being marked *READY* for execution (Lines 12-14).

---

**Listing 1** Fork rule implementation using the VIATRA Transformation Language.

---

```

1 val forkableRule = createRule.name("forkable").precondition(forkable)
2   action [
3     logger.debug(String.format("Forking token %s at %s.", token, fork))
4     // de-activate parent
5     token.abstract = true
6     // create sub-tokens
7     for (ctrlOut : fork.controlOut) {
8       val newToken = EnactmentFactory.eINSTANCE.createToken
9       newToken.subTokenOf = token
10      enactment.token.add(newToken)
11      newToken.currentNode = ctrlOut.to
12      if (ctrlOut.to instanceof Activity) {
13        newToken.state = ActivityState:::READY
14      }
15    }
16  ]
17  .build
```

---



---

**Listing 2** Related model queries implemented using the VIATRA Transformation Language.

---

```

1 pattern forkable(fork: Fork, token: Token){
2   find tokenInNode(token, fork);
3 }
4
5 pattern tokenInNode(token : Token, node : Node){
6   find activeToken(token);
7   Token.currentNode(token, node);
8 }
9
10 pattern activeToken(token: Token){
11   Token.abstract(token, a);
12   a == false;
13 }
```

---

The *forkable* precondition of the transformation rule is shown in Listing 2. The model query refers to another model query called *tokenInNode* (using the *find* keyword) and binds that query to the given control node of type *Fork*. This means, the query engine will look for a token in a fork control node. The *tokenInNode* query looks for an *activeToken* in the current node. The *activeToken* pattern, then, finds an object of type *Token* which is not abstract.

## 5.4 On-line inconsistency management

In this section, we present **Contribution 6** of this thesis: a formalism for modeling complex engineering processes. The majority of this section has been published in [43].

We focus on detecting inconsistencies arising once the process is enacted, i.e. “running”. We aim to detect inconsistencies as *early* as possible by leveraging the interrelations of the properties, attributes and their relationships. To this end, a state-of-the-art symbolic solver is presented.

The process is augmented with consistency checks during the enactment, which can be viewed as special activities of the process. These activities are not shown in the original engineering process, as they do not carry relevant information from the engineering point of view.

The explicit modeling of attributes and constraints means that the information relevant to inconsistency management is being conceptually “lifted” from the models containing those attributes and constraints. (Although, they are still present in the models themselves.) This modeling step may be expensive for larger processes, but it has to be done only once for a process. Attributes and constraints operate on multiple meta-levels, which enables reasoning about *capabilities*Section 5.1.2.4 of certain modeling formalisms. This, due to the strongly typed process model, provides additional vital information regarding potential inconsistencies along the process. This combination of modeling paradigms (i.e. multi-level, multi-abstraction, process-oriented) is novel in the state-of-the-art.

**Running example** In this section, we use another aspect of the AGV case study used throughout this chapter.

The engineering process of the AGV needs to determine the sizing of the different components (motors, battery, platform) and tune the controller. The process requires a collaboration between different stakeholders and their domain-specific engineering tools, such as CAD tools for platform design, Simulink and Virtual.Lab Motion for multi-body simulations, AMESim for multi-physical simulations during drivetrain design. The motor and battery selection activities use databases maintained in Excel files. Since these tools work with different modeling formalisms, reasoning over the consistency of the system as a whole properties poses a complex problem. By explicitly modeling attributes and constraints of the system and associating them with the engineering activities, the engineering process can be augmented with automated consistency monitoring.

The total mass of the AGV ( $m_T$ ) is a sum of the mass of the battery ( $m_B$ ), the mass of the

motor ( $m_M$ ) and the mass of the platform ( $m_P$ ):

$$m_T = m_P + m_M + m_B. \quad (5.1)$$

During the engineering process, and specifically: during the requirements analysis, constraints are applied on the attributes:

$$\begin{aligned} m_T &\leq 150 [kg], \\ m_P &\leq 100 [kg], \\ m_M &\leq 50 [kg], \\ m_B &\leq 10 [kg]. \end{aligned} \quad (5.2)$$

These constraints must be respected throughout the whole process, otherwise the model of the system becomes inconsistent.

Additionally, all the masses must be positive numbers, as constrained by the laws of physics.

$$mass > 0 [kg]. \quad (5.3)$$

Obviously, the notion of masses specific to the system, i.e.  $m_T$ ,  $m_B$ ,  $m_M$  and  $m_P$ , are of a different nature than the general *mass* concept, as *mass* is not related to a specific part of the system, but is more abstract. The concept of *mass*, in fact, can be viewed as *type* to the system-specific masses:  $m_T$ ,  $m_B$ ,  $m_M$  and  $m_P$  are all masses and therefore, a constraint on *mass* imposes a constraint on each system-specific mass. This means the constraint in Equation 5.3 must hold for each system-specific mass.

**Step 1** A platform is selected with a mass of 100 kg. ( $m_P = 100 [kg]$ )

**Step 2** A motor is selected with a mass of 50 kg. ( $m_M = 50 [kg]$ )

**Step 3** A battery is selected with a mass of 10 kg. ( $m_B = 10 [kg]$ )

At this point, an inconsistency can be detected. Even though the selected components satisfy their respective constraints imposed in Equation 5.2, the total mass now becomes 160 kgs (due to Equation 5.1), which leads to a violation of a constraint in Equation 5.2 and therefore: the design is considered inconsistent.

Factoring in Equation 5.3, however, allows an earlier detection of inconsistencies. Already in *Step 2*, Equation 5.1 can be rewritten as follows:

$$m_T = 150 + m_B.$$

Since Equation 5.3 holds for any mass, and consequently for  $m_B$  as well, it can be inferred that after selecting a battery (and thus filling in  $m_B$  in this equation), the total mass will be greater than 150, thus violating the constraint in Equation 5.2.

Such an *early* detection of inconsistencies may save significant costs in the specific engineering process, because it reduces the amount of iterations over complex engineering activities if the design is detected to be inconsistent. Early detection of inconsistencies requires (i) reasoning over constraints on different meta-levels (in this case: factoring Equation 5.3 into Equation 5.1); and (ii) efficient constraint solving algorithms.

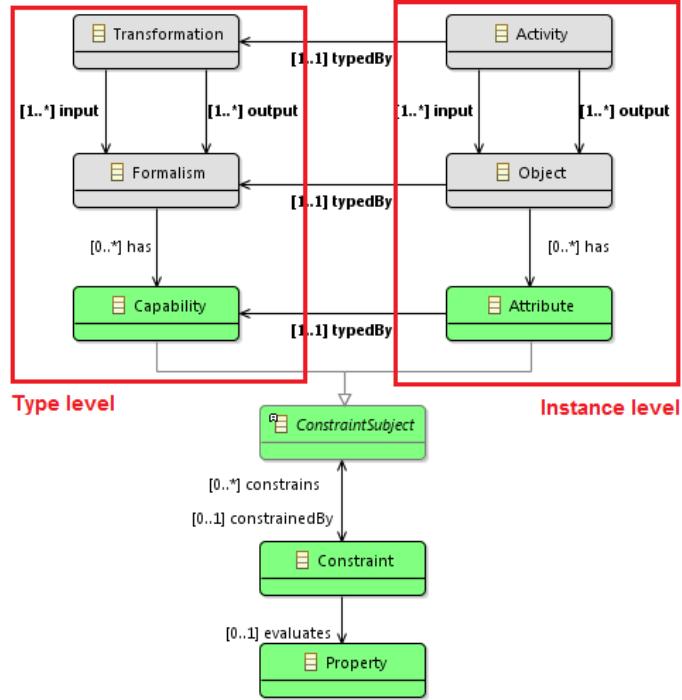


Figure 5.28: Excerpt from the extended FTG+PM metamodel supporting early online detection of inconsistencies.

### 5.4.1 Modeling support

The strong type system of the FTG+PM formalism fits well with the problem sketched in Section 5.4 as it supports reasoning on different meta-levels. To enable modeling the problem at hand, we make use of the *Attribute* type, and extend the FTG+PM formalism by its meta-type: the *Capability* (Figure 5.28).

As discussed in Section 5.4, the concept of *mass* is different from the concept of the masses specific to the system:  $m_T, m_B, m_M$  and  $m_P$  are related to the notion of *mass* by a typing relationship. In our framework, we call these “meta-attributes” *capabilities*

**Definition 2** (Capability). *A capability of a formalism expresses the ability of a model, corresponding to the formalism, to reason about attributes.  $\forall \theta \in \Theta \exists \gamma \in \Gamma : \text{typeof}(\theta) = \gamma$ .*

In the running example, Matlab is used for defining the (simplified) mechanical model of the AGV. The Matlab language, in this sense, is able to reason about masses. (Although, mass-like attributes are just ordinary data structures from Matlab’s point of view.)

To make use of attributes and capabilities for consistency management purposes, *constraints* are used to define validity rules over capabilities and attributes.

**Definition 3** (Constraint). *A constraint defines the desired characteristics of the system, i.e. it is a selection of an interval over the domain of the previously obtained attribute.*

Constraints are special types of relationships, presented in Section 5.1.2.2. Constraints are always of a level L3. In a typical engineering process, algebraic (Equation 5.1), arithmetical (Equations 5.2 and 5.3) and logical formulas are used as constraints. As shown in Figure 5.28, constraints can be applied on both the PM and the FTG side.

The design of the system is considered to be consistent at a given point of the process iff there are no violated constraints.

## 5.4.2 Modeling the running example

After having defined the core concepts, we use our prototype tool to model the attributes, capabilities and constraints of the engineering process. The tool provides a visual interface for modeling. It was built on top of the Eclipse platform, implemented using the Sirius framework [59], and it is available as an open-source software.

### 5.4.2.1 Attributes and their constraints

Figure 5.29 shows an excerpt from the full model of the example, with the attributes of the running example and their constraints modeled.

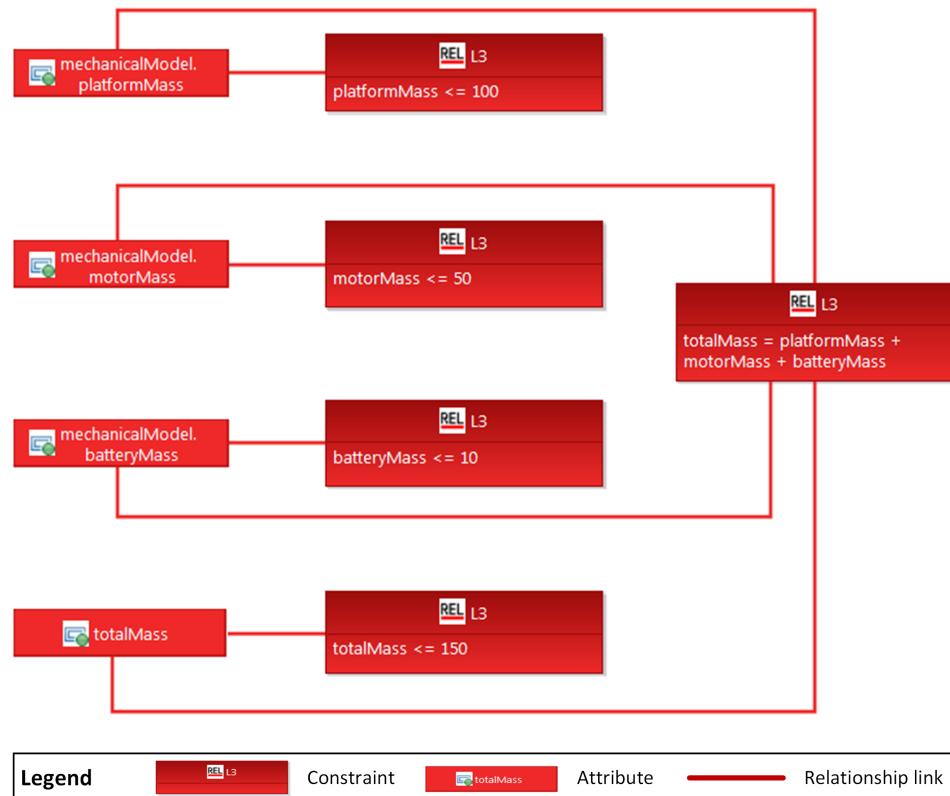


Figure 5.29: Attributes and constraints.

Attributes are denoted by light red rectangles, and constraints by darker red rectangles. There are four attributes in the figure, one for each of the masses in Section 5.4. Apart from the *totalMass*, the three other masses are persisted in the *mechanicalModel*, as shown by the prefix in the names of the attributes. The total mass is not persisted in the mechanical model, but it is a result of an aggregation of the other three masses (Equation 5.1). This equation is captured in the rightmost constraint, as shown by the formula. The other four constraints correspond to the four sub-equations of Equation 5.2.

The header of the constraint contains its level of precision. In this example, all of the constraints are of level L3. As defined in [42], the level of precision reflects what information a constraint carries:

- L1: the **fact of influence** is known, its extent is not;
- L2: **sensitivity information** between two values is expressed, e.g. by Forrester system dynamics [72];
- L3: the constraint can be expressed using an exact **mathematical relationship**.

In this work, we assume L3 relationships, but our technique can adapted to deal with lower levels of precision as well.

#### 5.4.2.2 Capabilities and their constraints

Figure 5.30 shows an excerpt from the full model of the example, with the *mass* capability, its constraint, alongside the related part of the FTG. *Matlab* is used as a formalism for defining the *mechanical model* of the system and has a capability of expressing *mass*. (That is, models being conform to the Matlab formalism, can have attributes of type *mass*.) Figure 5.30 shows how this aspect of the running example is modeled, with Equation 5.3 captured in a similar fashion as the other constraints were in Figure 5.29.

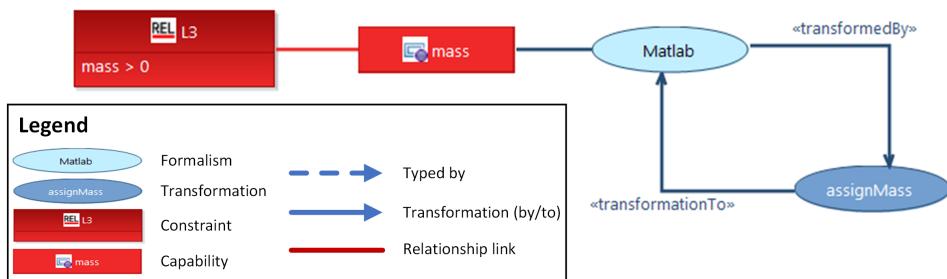


Figure 5.30: A capability and its constraint in the FTG.

The evaluation of such constraints, however, differs from the ones shown in Figure 5.29, as the constraints on *mass* are stemming from the universal laws of physics, while  $m_T$ ,  $m_P$ ,  $m_B$ ,  $m_M$  are specific to the system. To evaluate constraints of capabilities, we use the following rule.

**Definition 4** (Evaluation of capability constraints). *Any constraint applied on a capability imposes a constraint on every attribute typed by that capability.*

This means, that based on the typing relationship between the *mass* capability and the system-specific masses  $m_T, m_P, m_B, m_M$ , Equation 5.2 can be unfolded as follows:

$$\begin{aligned} 0 [kg] < m_T \leq 150 [kg], \\ 0 [kg] < m_P \leq 100 [kg], \\ 0 [kg] < m_M \leq 50 [kg], \\ 0 [kg] < m_B \leq 10 [kg]. \end{aligned} \quad (5.4)$$

#### 5.4.2.3 Properties

As presented in [203] and [107], ontologies can be efficient enablers for inconsistency management in heterogeneous settings. During the translation of requirements to view-specific properties, each stakeholder keeps in mind certain domain properties, i.e. ontological properties. For example, an electrical engineer implicitly thinks about the capacity of the battery, a mechanical engineer reasons about how a battery would fit the frame of the AGV. Due to overlap in requirements, some ontological properties will be shared and/or will influence each other such that the related view-specific properties will be shared or influenced as well. Our framework allows defining property satisfaction relationships in terms of attributes. The satisfaction of a property can be inferred by checking the single constraints imposed on the specific attributes.

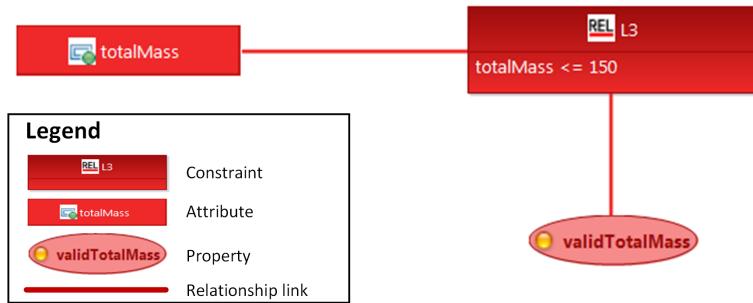


Figure 5.31: Excerpt from the example: the property *validMass*.

Figure 5.31 shows the property *validMass*. The satisfaction of the property is evaluated from the *totalMass* attribute, using the same constraint as the one shown in Figure 5.29 (i.e. the constraint on attribute  $m_T$  in Equation 5.4), while the bottom right element, labeled with the name of the property, holds the boolean value of the satisfaction relationship.

This notion of properties allows various scenarios aiding inconsistency management, such as reusing domain knowledge from existing domain ontologies [68], using ontological

reasoning [203] in conjunction with our techniques, and using contract-based design [159] to aim co-design scenarios, i.e. parallel branches of the engineering process.

#### 5.4.2.4 Putting it all together: the process

Figure 5.32 shows the final model of the running example. In the middle, the yellow rectangles denote the activities of the PM with control flows in between them denoting the precedence relationship between the activities. On the right side, the attributes and constraints are shown. Activities and attributes are linked by *intents*, which express the purpose of the activity accessing a given attribute. The first three activities access attributes in order to *modify* them, while the last activity attempts to resolve a constraint wrt *totalMass*. Other types of intents include: reading the value of an attribute, imposing a constraint, locking/releasing an attribute in a parallel process branch, etc. This latter step is built into the process as an actual engineering step, but as shown later, in case of an inconsistency, the consistency monitoring service can stop the process before this point. In our previous work [41] we used *read-modify* pairs of intents to identify potential inconsistencies at the optimization phase. In this work, however, we leverage the notion of intents at run-time to narrow the scope of the consistency checking algorithm, i.e. to consider only the attributes which have been explicitly linked with an activity using a *modify* intent.

On the left side, the FTG and the only associated capability is shown. The typing relationships correspond to Figure 5.28:

- the mechanical model in the PM is an Object and it is typed by the Formalism *Matlab* in the FTG;
- the Activities are typed by the transformation *assignMass*;
- finally, the *mass* capability types all the masses on the right side, but this relationship is not visualized in the graphical view. (It is shown in a property view of the tool, however.)

Masses are assigned to the design when the respective activities are executed. Conceptually, this assignment can be viewed as a transformation of the model, and as such, the actual transformation logic is captured in the transformation typing the activities. In this case, *assignMass* holds the specification of the transformation. The activities operate on models. To check the consistency of the attributes, the attribute values are obtained by querying the appropriate models, i.e. the ones the specific attributes are persisted in. As Figure 5.32 shows, the name of the attribute is prefixed with the name of the model persisting the attribute. The first three attributes are all persisted in the *mechanicalModel*, which is a *Matlab* type of a model. Using this information, the querying is executed in the background by our tool via the Matlab API, without requiring the user to submit any extra information for this.

All these values come from the specifications, previously inferred from the requirements by the stakeholders. Depending on the scenario, the values may already have been persisted in the respective domain models and lifted to this level of the process during the specification of the process (i.e. queried from the domain model); or a stakeholder may have used the process model to specify the parameters of model manipulations.

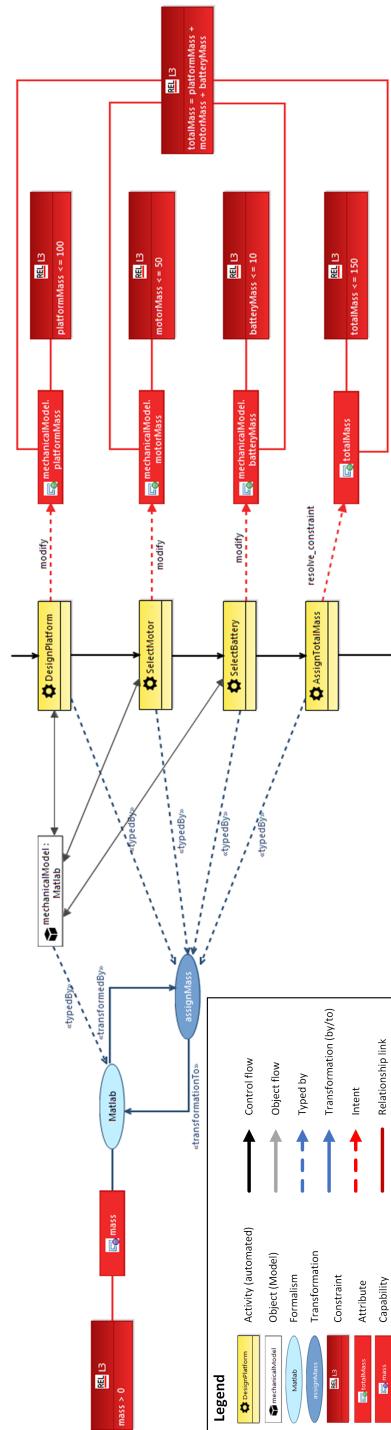


Figure 5.32: The FTG+PM based process model with the capabilities (left) and the attributes (right).

### 5.4.3 Early detection of inconsistencies

Now we discuss how the detection of inconsistencies is carried out while the process is being enacted.

#### 5.4.3.1 Algorithm for early inconsistency detection

The early detection of inconsistencies requires computing the satisfiability of the system of constraints at certain points of the process. These computations are carried out on each *Step* in the process, based on Algorithm 2.

---

#### Algorithm 2 Handling attribute modifications.

---

```

1: procedure STEP(token, nextActivity)
2:   token.currentActivity  $\leftarrow$  nextActivity                                 $\triangleright$  Move the token to the next activity
3:   for all i:Intent, a:Attribute: i(token.currentActivity, modify, a, v) do
4:     UPDATEATTRIBUTEVALUE(a, v)                                          $\triangleright$  Assign value v to attribute a
5:   end for
6: end procedure

```

---

On each *Step* in the process, the token is moved to the next activity (Line 2). As discussed in Section 5.1.2.3, *intents* between activities and attributes help identifying the cases when an activity modifies the value of a property. For each of such intents (Lines 3-5), the attribute is updated and the change is propagated through the whole system of constraints. This latter step is being taken care of by Algorithm 3.

## Symbolic computation of constraints

The updates to the system of constraints require introducing the new values of attributes and computing whether the constraints can be still satisfied later on in the process or not. Such a computation requires factoring in the potential impacts of the *future* attribute changes (explicitly modeled in the process). To execute these computations, we opted for the techniques of *symbolic computation*. Our main concern is the maintenance of a system of constraints by gradually simplifying them as attributes get updated, to the point, where contradictions appear in the equations, i.e. the set of potential solutions is empty, thus denoting an inconsistency in the system design. Alternative approaches include simulation of the process and abstract interpretation.

Algorithm 3 shows the steps taken in our symbolic computation approach. The algorithm is invoked by Algorithm 2 with the name and the new value of the attribute to be updated passed along as parameters. In *Phase 1* of the algorithm, the attribute-value assignment is translated to an equality constraint and added to the system of constraints (Line 2). In *Phase 2*, the algorithm propagates this change and attempts to simplify every constraint. This is achieved by iterating through the system of constraints (Lines 3-4) and factoring equation constraints (Line 5-6) into the rest of the constraints by trying to solve (simplify) the constraint (Line 7).

We use the SymPy [176] symbolic mathematics library to solve/simplify the constraints, thus the semantics is provided by the library. Constraints imposed by capabilities are calculated based on Definition 4 and applied on attributes.

Finally, in case an empty set is produced as a set of potential solutions for a constraint, we interpret it as an inconsistency and notify the user about this fact.

---

**Algorithm 3** Maintenance of the system of constraints.

---

```

1: procedure UPDATEATTRIBUTEVALUE(attribute, value)
   Phase 1 – Impose a new constraint with equality
2:   model.constraints  $\leftarrow$  Eq(attribute, value)
   Phase 2 – Propagation and simplification: substitute equality constraints into the rest of the constraints
3:   for all constraint1 in model.constraints do
4:     for all constraint2 in model.constraints do
5:       if constraint2 is Eq then
6:         constraint1  $\leftarrow$  constraint2
7:         solution = solve(constraint1)                                 $\triangleright$  Try to solve the constraint
8:         if solution =  $\emptyset$  then
9:           notify inconsistency
10:          end if
11:        end if
12:      end for
13:    end for
14: end procedure

```

---

When an inconsistency is detected, the process is halted and cannot proceed until the inconsistency is not fixed. Resolving inconsistencies is outside the scope of the paper. A simple undo/redo functionality is provided by the framework, but more detailed research have been carried out by other authors, e.g. [128], [65], [60], or [6].

#### 5.4.3.2 Execution of the running example

We follow the process in Figure 5.32. During the execution, Equations 5.1 and 5.2 are maintained: whenever a value is assigned to an attribute present in one of the equations, the equations are simplified with that attribute. This means

- substituting the newly assigned value of the attribute to every occurrence of the attribute in every equation; and
- removing constraints without free attributes.

**Step 1: The platform mass is set to 100 kgs.** Activity *DesignPlatform* is executed and the mass of the platform is set in the mechanical model. Since the attribute is persisted in a Matlab model, the model is queried via the Matlab API for the value of the *platformMass* variable. The consistency manager uses this information to update the constraints with. The related constraint of Equation 5.2 ( $0 \text{ [kg]} < m_P \leq 100 \text{ [kg]}$ ) is

satisfied, and therefore the system can be simplified with it:

$$m_T = 100 + m_M + m_B [kg]$$

$$\begin{aligned} 0 [kg] < m_T &\leq 150 [kg] \\ 0 [kg] < m_M &\leq 50 [kg] \\ 0 [kg] &< m_B \leq 10 [kg] \end{aligned}$$

Solving the constraints for  $m_T$  results in a non-empty set of solutions:

$$100 [kg] < m_T \leq 150 [kg]$$

No inconsistency is detected, the process proceeds.

**Step 2: The motor mass is set to 50 kgs.** The *SelectMotor* activity is executed which sets the mass of the motor to 50 kgs.

$$m_T = 150 + m_B [kg]$$

$$\begin{aligned} 0 [kg] < m_T &\leq 150 [kg] \\ 0 [kg] &< m_B \leq 10 [kg] \end{aligned}$$

At this point, Algorithm 3 detects the inconsistency sketched in Section 5.4. Solving the constraints for  $m_T$  results in an empty set of solutions:

$$150 [kg] < m_T \leq 150 [kg]$$

Since  $0 < m_B$ , it can be inferred, that after executing the next activity of the process,  $m_T > 150$  will hold, which violates the constraint on the total mass. The process is halted and a notification is raised to the user to resolve the inconsistency.

#### 5.4.4 Discussion

As demonstrated, factoring ontological knowledge into the requirements of the system may shed light to additional constraints viable for early detection of potential inconsistent states of the system design. The main advantage of this approach is the support for such scenarios by uniformly handling instance- and meta-level constraints. As highlighted in the running example, the advantages of such an early detection approach are visible already in very simplistic cases. The gain realized in real engineering processes can obviously be much higher, when the execution of resource and cost demanding activities can be prevented by the early detection of inconsistencies.

The proposed modeling formalism enables lifting information relevant to inconsistency management purposes regarding the given process. Explicit modeling of such information is an enabler of improving the quality and efficiency of the process once enacted. Although

the thorough modeling requires significant efforts from the stakeholders, it is needed to be done only once, before the actual design of the system commences. Such a front-loaded approach can be typically expected from companies on CMMI levels 3 and above [35]. As a consequence, our approach suits best the domain of complex heterogeneous systems, where the costs of dealing with inconsistencies is often in a different order of magnitude than the costs of modeling and optimizing the process.

## 5.5 Translating process models to DEVS for performance evaluation

In this section, we present **Contribution 7** of this thesis: a formalism for modeling complex engineering processes. The majority of this section has been published in [46].

In this section, we provide an advanced simulation technique for the off-line inconsistency management technique presented in Section 5.2. The more simple performance evaluation techniques presented in Section 5.2.3.5 can be easily replaced by the technique presented in this section to achieve more precise results.

We provide an automated translation from engineering process models to a formalism appropriate for the simulation of performance characteristics of the process. To show that our approach is applicable to all possible process models, we base ourselves on the essential process/workflow patterns, previously identified by [187]. Each of these patterns introduces essential concepts of a process model, thereby providing us with a list of elements that should be supported. While our approach is complete, only the most common constructs are presented here due to space restrictions.

Our mapping is furthermore specialized in two dimensions: (i) the stochastic nature of process execution; and (ii) the constraints imposed by the resource demands of the process.

In our approach, instead of modeling with DEVS, we allow the high-level, more appropriate process formalisms to be used and support mapping to DEVS in the background.

### Performance of processes

To evaluate the performance of a process model we assume a metric space  $(W, d)$  with  $W$  being the set of processes and  $d$  being a metric on  $W$ , i.e. a function  $d : W \times W \rightarrow \mathbb{R}$ , [33], used for comparing two arbitrary process models  $w_i, w_j \in W$ . Because of the domain of  $\mathbb{R}$ , the set of performance metrics constitutes an ordered set  $(\mathbb{R}, \leq)$ , i.e.  $\forall d_1, d_2 \in \mathbb{R} : d_1 \rightarrow d_2 \iff d_1 \leq d_2$ . In this paper we use the simple metric of *transit time*  $t$  as a quantified performance metric. Given two transit time metrics  $t_1$  and  $t_2$ , the better performance is associated with the lower value, following the intuitive interpretation of the time-based performance metrics. Apart from the transit time, typical performance metrics in business and engineering processes include other time-based metrics (e.g., time-to-market, queueing time) or monetary value based metrics (e.g., monetary emerging costs); and more complex metrics (e.g., resource utilization).

## Simulation by DEVS

To obtain the performance metric of transit time, we translate processes to DEVS [216]. Given its ability to support queueing systems and performance modeling [216], we determine DEVS to be an appropriate formalism for the performance analysis of process models. DEVS is a discrete-event formalism, consisting of Atomic DEVS models (defining behaviour) and Coupled DEVS models (defining structure). Atomic DEVS models are defined as an 8-tuple  $\langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$  [199, 201]. Coupled DEVS models are defined as a 7-tuple  $\langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle$ . To keep our approach as general as possible, we do not tie ourselves to one particular process modeling formalism, but rather, we provide a mapping of each of the original 20 *van der Aalst* process/workflow patterns [187] to DEVS models. Given these patterns, every process can be mapped on to DEVS and simulated.

## Modeling the stochastic nature of processes

To obtain meaningful results, the process model must be calibrated with activity execution times. These execution times are, however, often based on distributions, as some variation will be present in this value. For this reason, the time taken by activities is best defined through a distribution, thereby having a further impact on the complexity of the time taken by the process. Additionally, this distribution likely evolves over time, depending on the number of iterations that have previously occurred. A typical example in model-driven engineering processes is the activity of creating a model, which gradually takes less time as the process iterates through it, since the modeler does not have to restart from scratch. Another stochastic element is the decision: the activation of the outgoing branches depends on a condition that can only be evaluated at execution time. The choice is sampled from a distribution, which is again likely to evolve over time. For example, the first time a model is simulated for a property, it is likely that there are still unsatisfied elements in the model, but this chance gradually decreases as the check is more often performed.

## Modeling resource constraints

An additional facet of processes are the constraints imposed by the resources available for the specific activities. The execution of an activity demands specific amounts of specific resources types, such as a CPU core, an engineer, a software license, etc. Depending on the other (concurrently) execution activities, these resources might not be available, meaning that the activity cannot start execution yet. By introducing these limitations in our simulation, it becomes possible to determine the effect of obtaining additional resources (e.g., hiring an additional employee), or releasing it. It thus becomes possible to optimize the available resources, while taking into account the impact on process enactment time.

## Implementation

Our approach is implemented in the Modelverse [200], our MPM environment. The Modelverse presents the users with a process modeling environment, where the mapping to

and simulation of DEVS models remains completely hidden.

## Running example

To illustrate our approach, we use the running example shown in Figure 5.33.

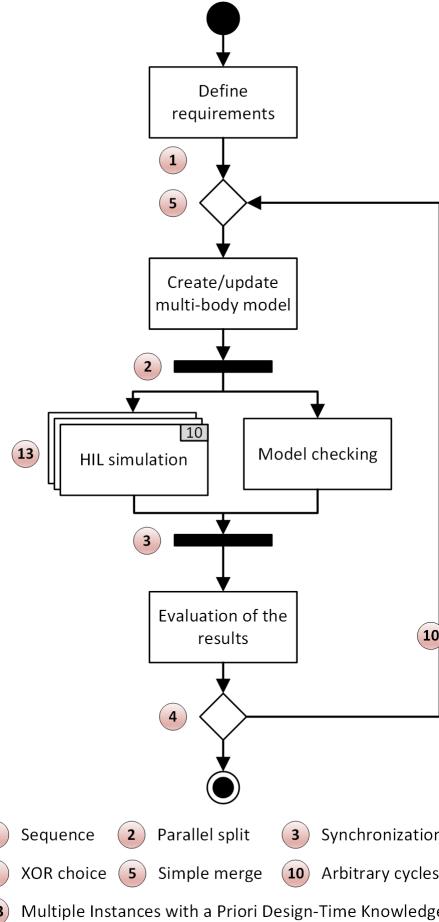


Figure 5.33: The process of selecting the optimal hardware component.

The process depicts the simplified engineering scenario of selecting the appropriate configuration of a hardware module through modeling and hardware-in-the-loop (HIL) simulations. This is commonly done for example for Automated Guided Vehicles (AGVs) [41]. The process is augmented with the identifiers of the relevant *van der Aalst* process/workflow patterns at the appropriate locations. First, the *requirements are defined*. Subsequently, the *Multi-body model is being created*, or updated, depending on the iteration. The process is then split into two parallel branches. In one branch, ten independent *HIL Simulations* are carried out in a parallel way to measure the physical attributes of the system and obtain statistical characteristics, such as the mean and variance. In the other branch, the model

of the system undergoes formal *model checking* to verify the satisfaction of the safety requirements. Both branches require a significant amount of CPU time for the involved computations. After the synchronization of the branches, results are evaluated. If requirements are met, the process finishes; otherwise another iteration of engineering work is enforced.

Although in our running example we only focused on the control flow of the process, the approach presented in this paper fits well with our previously presented approach, based on the FTG+PM formalism. [121] The rich semantics of the FTG+PM formalism allow reasoning not just over the syntactic characteristics of input data, but also the semantic characteristics, resources and costs.

### 5.5.1 Translating processes to DEVS

In order to simulate the enactment of the process model using DEVS, an automated translation between the two is necessary. Since DEVS constitutes Atomic and Coupled DEVS models, the translation consists of two parts. The first part is the Atomic DEVS models, defining behaviour. As the behaviour of the process nodes is fixed, these can actually be hardcoded in Atomic DEVS model. For example, the behaviour of a Parallel Split is independent of how it is used in the process. To tackle this part of the translation, we create a library of Atomic DEVS models of which the elements can be parameterized with runtime information (e.g., number of incoming links for Synchronization). The second part is the Coupled DEVS model, defining structure. Given the set of Atomic DEVS building blocks, we must translate each process node to the equivalent Atomic DEVS model, and create the correct links between the two of them. For example, the sequence of two activities is translated to a connection in the Coupled DEVS model. Only this part of the translation is specific to the process model. This is summarized in the FTG+PM [121] model shown in Figure 5.34. At the left-hand side, the Formalism Transformation Graph (FTG) presents the different formalisms and activities between them. At the right-hand side, the Process Model (PM) presents the process as an instance of the FTG, showing the order in which activities are invoked and how the models are used.

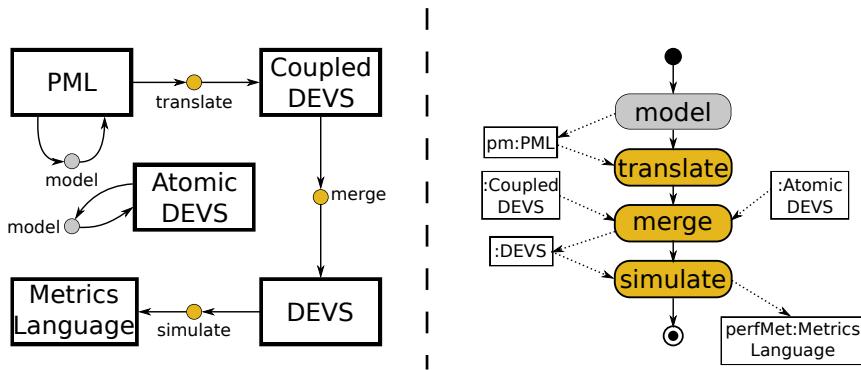


Figure 5.34: FTG+PM model of our approach.

### 5.5.1.1 DEVS Library

The first step is to create a library of Atomic DEVS models, one for each process node presented by [187]. Due to space restrictions, we only present the patterns used in our running example. All elements have a similar structure, with an input and output port termed “resource\_in” and “resource\_out”, respectively. Over these ports, there is a control token that passes from node to node. As soon as a node receives the control token, it knows that the activity before it has terminated. Most process nodes subsequently pass on the token in one way or the other (e.g., split it or wait for multiple). Further details are given next for each type of process node.

1. *Activity*. The activity takes in the control token and passes it on after some time. While not a pattern, it is the basic building block of a process, and could be considered as the Sequence pattern (Pattern 1). In DEVS, this can simply be modeled as shown in Equation (5.5). Essentially, the atomic DEVS model is either active (i.e., currently executing) or inactive (i.e., waiting for control). When the model is active, it outputs the token after some time  $t_{process}$ , otherwise it passivates.
2. *Parallel Split*. The parallel split (Pattern 2) merely duplicates a token instantaneously, thereby effectively starting concurrent branches. It is therefore identical to the activity, but instead of taking  $t_{process}$  time to process the token, it happens instantaneously. The control token is put on the output port, to which all subsequent nodes are connected. As such, the splitting of the token happens automatically. The DEVS specification is not repeated, as it is identical to Equation (5.5), but with  $t_{process}$  set to zero.
3. *Synchronization*. Synchronization (Pattern 3) has to wait for all incoming control flows to send a token. From the patterns definition, we know that only a single control token is sent on each branch (as the input places are guaranteed to be safe). Therefore, we can simply keep a counter, counting how many incoming tokens we need to receive, before we forward it ourselves. This counter is initialized with the number of incoming branches, which is specific to the process. This is shown in Equation (5.6).

$$\text{Activity} = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5.5)$$

$$\begin{aligned} X &= \{\text{ControlToken}\} \\ Y &= \{\text{ControlToken}\} \\ S &= \{\text{active}, \text{inactive}\} \\ q_{init} &= (\text{inactive}, 0.0) \\ \delta_{int} &= \{\text{active} \rightarrow \text{inactive}\} \\ \delta_{ext} &= \{((\text{inactive}, \_), \text{ControlToken}) \rightarrow \text{active}\} \\ \lambda &= \{\text{active} \rightarrow \text{ControlToken}\} \\ ta &= \{\text{active} \rightarrow t_{process}, \text{inactive} \rightarrow \infty\} \end{aligned}$$

$$\text{Synchronization} = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5.6)$$

$$\begin{aligned} X &= \{\text{ControlToken}\} \\ Y &= \{\text{ControlToken}\} \\ S &= \mathbb{N} \cup \{\text{inactive}\} \\ q_{init} &= (\text{inactive}, 0.0) \\ \delta_{int} &= \{0 \rightarrow \text{inactive}\} \\ \delta_{ext} &= \{((i \in \mathbb{N}, \_), \text{ControlToken}) \rightarrow i - 1\} \\ \lambda &= \{0 \rightarrow \text{ControlToken}\} \\ ta &= \{0 \rightarrow 0.0, \mathbb{N} \setminus \{0\} \rightarrow \infty, \text{inactive} \rightarrow \infty\} \end{aligned}$$

4. *Exclusive Choice.* The exclusive choice (Pattern 4) makes a decision depending on some condition, thereby passing the control token to only a single branch. In the context of our process, we need to make it such that the decision varies between different invocations, as otherwise the process might become stuck in a loop. Later in this paper, we make this choice probabilistic, thereby circumventing these problems.
5. *Simple Merge.* The simple merge (Pattern 5) merges different branches again as soon as a single one of them proceeds. This is commonly used after an exclusive choice, when the conditional part is finished, but can also be used in other scenarios where branches must merge. In contrast to synchronization, the simple merge continues for each token that comes in and does not wait for any other input. This definition is again similar to that of an activity, as it essentially takes in a control token and immediately passes it on to the next node. As such, the definition is identical to Equation (5.5), but with  $t_{process}$  set to zero.
6. *Multiple Instances with a priori Design-Time Knowledge.* The multiple instance process node (Pattern 13) is a more complex node, which spawns several instances of a single activity concurrently. The number of instances is defined at design time, making it possible to know beforehand how many instances there will be. Nonetheless, this information is still unknown in our Atomic DEVS model, as the library blocks are generic, and must therefore be taken in as a parameter. Control progresses as soon as all spawned threads are finished, meaning that we operate synchronously. For the moment, this definition is again the same as the activity, shown in Equation (5.5), as nothing prevents us from spawning these elements concurrently immediately. Indeed, given that each instance takes  $t_{process}$  amount of time to process, but they can all run concurrently, the total time taken is again  $t_{process}$ .
7. *Initial.* While not an actual pattern, each process model needs exactly one initial node, to start the process. This has its own specific DEVS model, which starts with the control token and immediately hands it over to the next node. The DEVS model is shown in Equation (5.7). Essentially, the model immediately sends out the token and then passivates in the inactive state.

$$Initial = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5.7)$$

$$\begin{aligned} X &= \{\} \\ Y &= \{ControlToken\} \\ S &= \{active, inactive\} \\ q_{init} &= (active, 0.0) \\ \delta_{int} &= \{0 \rightarrow inactive\} \\ \delta_{ext} &= \{\} \\ \lambda &= \{active \rightarrow ControlToken\} \\ ta &= \{active \rightarrow 0.0, inactive \rightarrow \infty\} \end{aligned}$$

$$Finish = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5.8)$$

$$\begin{aligned} X &= \{ControlToken\} \\ Y &= \{\} \\ S &= \mathbb{R} \cup \{inactive\} \\ q_{init} &= (inactive, 0.0) \\ \delta_{int} &= \{\} \\ \delta_{ext} &= \{((inactive, e), ControlToken) \rightarrow e\} \\ \lambda &= \{\} \\ ta &= \{* \rightarrow \infty\} \end{aligned}$$

8. *Finish*. Similar to the initial node, the finish node is not actually a pattern, although it is required to indicate that the process has finished. This again has its own DEVS model, which starts in a passivated state. As soon as the control token arrives, the elapsed time is stored in the state, as it will be used in the simulation termination condition later on. This elapsed time is the time taken by the process, as no other events ever arrive in this model. The DEVS model is shown in Equation (5.8).

### 5.5.1.2 Mapping through model transformations

With all Atomic DEVS models defined, the next step is to replicate the original process semantics with them. For this, we will make instances of the DEVS library elements for each of the process nodes, and copy the relevant connections between them.

Model transformations are ideally suited for this operation, as they allow for pattern detection and replacement. A model transformation defines a left-hand side (LHS), right-hand side (RHS), and negative application condition (NAC). For each application of the

transformation, a match for the pattern in the LHS is searched in the model. When found, the match is replaced with the RHS if simultaneously the NAC is not matched. Model transformations generate traceability links, which store the origin of a specific construct in the target. This can be used subsequently for matching, or later on for debugging. For example, each process node is linked to an instance of the activity Atomic DEVS model through such a traceability link. These links are later on used to find translated elements: with all nodes matched and traceability information stored, linking the control flow edges becomes trivial.

An example transformation rule for an activity is shown in Figure 5.35. Here, we match on an abstract superclass (*Node*) and the related DEVS Atomic DEVS models, through the traceability links (links 6 and 7).

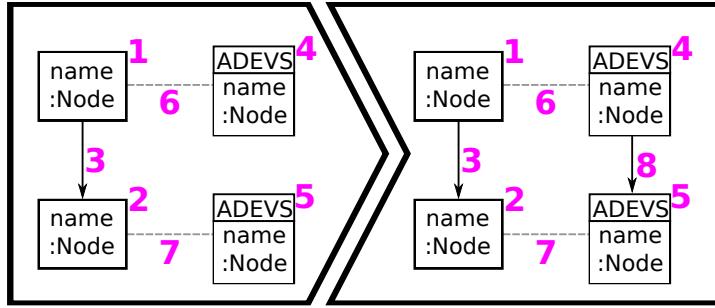


Figure 5.35: Transformation rule for control flow links.

### 5.5.1.3 Implementation

The approach was implemented in the Modelverse [200], our Multi-Paradigm Modeling (MPM) environment. When enacting the FTG+PM model shown before, the Modelverse automatically loads the appropriate formalism (Process modeling) and allows users to model with it. When users are content with the process model, enactment continues with the (automated) mapping to a Coupled DEVS model. This is then combined with the Atomic DEVS model library created beforehand to create a complete DEVS model. This DEVS model is serialized and sent to PythonPDEVS [198], our DEVS simulator, through the use of external services [195]. PythonPDEVS simulates the model and returns the simulation results, which get stored in the Modelverse.

The implementation is disclosed in Listings 14-19 of Appendix C. The coupled DEVS model of the running example is disclosed in Listings 20-21 of Appendix C.

## 5.5.2 Calibration of the process models

To evaluate the performance of the workflow, we augment its structure with multiple stochastic parameters, capturing the dynamics of the process. As Figure 5.36 shows, each activity is augmented with an *estimated execution time* (blue), and the characterization of the *evolution of the execution time* (red) over multiple iterations. Each activity is augmented with an *estimated execution time*, and the characterization of the *evolution of*

the execution time over multiple iterations. For clarity, we only show the evolution function for *create/update multi-body model*, as this is a constant function for all other activities. Finally, the outgoing branches of decision nodes are augmented with a quantitative *decision function*.

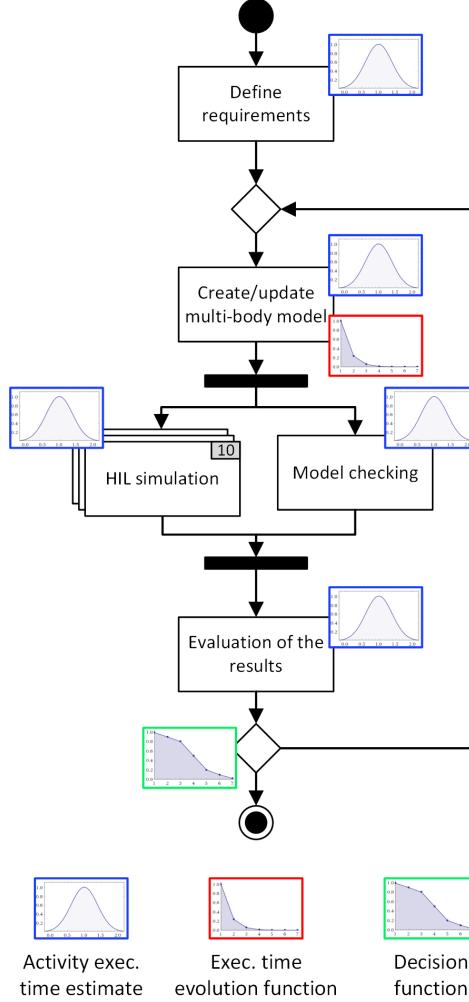


Figure 5.36: The augmented process.

The characteristics of the estimations and the resulting functions can be determined by looking into historical data, for example. At this point, our example values rely on rules of thumb taken from industrial partners. We now elaborate on the above parameters in detail.

1. *Estimated execution time of the activities.* The execution time of the activities is estimated by a normal (Gaussian) distribution. The distribution is set so that (i) its expected value represents the guess of the estimator, and (ii) its variance yields 80% of the estimations within a 20% error range. The latter characteristic is

achieved by setting the variance  $\sigma$  relative to mean  $\mu$ , resulting in the function of  $t(a) = N(\mu, 0.15625\mu)$ .

2. *Evolution of the execution time.* The previous estimates of the activity execution times might evolve during the subsequent iterations of the process. For example, an activity might cache previous results (e.g., in simulation) or use them as a basis (e.g., modeling from scratch vs. updating a model). To capture this aspect, we use an aggressive exponential function  $e^{-1/0.7i}$ , where  $i$  denotes the number of iterations. This factor is used to scale down the original estimates as follows:  $t(a, i) = t(a) * e^{-1/0.7i}$ . The function results the following scale factors for the first few iterations: 1.0, 0.2397, 0.05743, 0.01376, 0.00329, and 0.0008.

As shown in Figure 5.36, there is only one activity (*Create/update multi-body model*) with a decreasing execution time over the various iterations. Of course, the execution time might also remain constant or even increase.

3. *Decision function.* Deciding whether or not to re-iterate over a part of the process also happens with a given probability. A strong assumption against the process is its convergence towards a final solution. This also means that the chance of having to re-iterate over previously carried out activities decreases over time. We model this in a similar vein to the evolution of the execution time, again keeping track of the number of iterations. The function results the following probabilities of having to reiterate for the first few executions of the process: 0.99, 0.9, 0.8, 0.5, 0.2, 0.1.
4. *Incorporating the stochastic parameters in the DEVS models.* There are only three Atomic DEVS models that are influenced by these extensions. The first two, Activity and Multi-Instance, are actually similar, as they merely have to update their  $t_{process}$  from a constant to a sample from a random distribution. To cope with the evolution of the execution time, both types of process nodes keep a counter of how many times they have been executed. This is trivial to incorporate in the DEVS model and is therefore not shown explicitly. The third process node, the Decision node, similarly gets augmented with a counter which counts how many times it was executed. This counter is passed on to the distribution function when sampling for which output port to use.

### 5.5.3 Modeling resource constraints

Another process constraint is resource utilization. There are several types of resources, such as recurring costs (e.g., employees), one-time costs (e.g., software licenses), or a combination of both (e.g., CPUs). The constraints imposed by these resource requirements have a significant effect on the performance of the process. Indeed, if multiple activities are scheduled in parallel, but only enough resources are available for one (e.g., only one employee), these activities are effectively sequentially. As such, the inclusion of resources is an important consideration. For simplicity, we consider a generic “resource” in further discussions, although this could be any relevant resource.

### 5.5.3.1 Implications of Resources

In our running example, the use of resources becomes important in the concurrent part, where HIL simulation and model checking are done simultaneously. Both activities can be done automatically, meaning that relevant resources are for example software licenses (e.g., for the tool performing the simulation or model checking) and CPU cores (to execute the activities on). As only activity and multi-instance process nodes consume a non-trivial amount of resources, we focus on them next.

1. *Activity.* The first process node that is influenced, is the activity. Assuming that an activity has some resource requirements, which it needs throughout the complete duration of the activity. Should this not be the case, the activity can be split up in multiple distinct activities. As such, before starting the execution of the activity, we request all required resources. Only as soon as all resources are acquired, will the activity start executing. Upon termination of the activity, resources are released again.
2. *Multi-Instance.* The multi-instance node is similar to the activity node, except that multiple instances are spawned as soon as possible. Each instance is independent of the others, meaning that some instances could already be spawned, depending on resource availability. Instead of requesting all resources, the resources for a single instance are requested atomically. For example, if five instances have to be spawned, we make five individual requests, each requesting all resources required by the instance. If for example only two requests can be granted at that time, at least these two instances already acquire the resources and can start execution. After an instance has finished, its resources are already released. As soon as resources become available, additional instances are spawned.

### 5.5.3.2 Incorporating resources in the DEVS models

We now include these concepts in the Atomic DEVS models discussed before. As only two types of nodes need resources, we only consider these two. Additionally, the resource handler, being the entity which is responsible for the acquisition and release of resources, is modeled as well. To handle the communication between these nodes and the resource handler, all relevant nodes are augmented with a *resource\_out* and *resource\_in* port, all connected to the single resource handler.

1. *Activity.* For the activity, we merely add two new states to make a request for the resource and wait for the reply. Only upon a positive reply does the activity start its execution and its  $t_{process}$  starts to count down. When activity execution is finished, resources are released in the same output function where the control token was passed on. A DEVS specification is given in Equation (5.9).
2. *Multi-Instance.* The Multi-Instance node is a bit more difficult, as it considers multiple instances, all requiring the same resources. When the node gets the control token, it immediately requests all resources for all instances to be spawned. Upon receiving a positive reply from the resource handler, the  $t_{process}$  for one instance is sampled from the distribution and the countdown starts. These timers run in parallel, modeling the concurrent execution of a multi-instance node. Note that each instance

samples from the same distribution, although the actual times might vary. Instances that acquired their resources at the same time, might thus finish at different times. Only when all instances have finished execution will the model passivate again. A full DEVS specification is not presented in this paper due to space restrictions, although it can be found in our implementation.

$$\text{Activity} = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5.9)$$

$$\begin{aligned} X &= \{\text{ControlToken}, \text{Resources}\} \\ Y &= \{\text{ControlToken}, \text{Request}, \text{Release}\} \\ S &= \{\text{active}, \text{inactive}, \text{request}, \text{waiting}\} \\ q_{init} &= (\text{inactive}, 0.0) \\ \delta_{int} &= \{\text{active} \rightarrow \text{inactive}, \text{request} \rightarrow \text{waiting}\} \\ \delta_{ext} &= \{((\text{inactive}, \_), \text{ControlToken}) \rightarrow \text{request}, \\ &\quad ((\text{waiting}, \_), \text{Resources}) \rightarrow \text{active}\} \\ \lambda &= \{\text{active} \rightarrow [\text{ControlToken}, \text{Release}], \\ &\quad \text{request} \rightarrow [\text{Request}]\} \\ ta &= \{\text{active} \rightarrow t_{process}, \text{inactive} \rightarrow \infty, \\ &\quad \text{request} \rightarrow 0, \text{waiting} \rightarrow \infty\} \end{aligned}$$

$$\text{ResourceHandler} = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5.10)$$

$$\begin{aligned} X &= \{\text{Request}, \text{Release}\} \\ Y &= \{\text{Resources}\} \\ S &= [\text{Request}] \times \mathbb{N} \\ q_{init} &= ([], \text{resources}), 0.0 \\ \delta_{int} &= \{([R, r], i) \rightarrow ([R], i - 1)\} \\ \delta_{ext} &= \{(([R], i), e), \text{Request} \rightarrow ([R, \text{Request}], i), \\ &\quad ((([R], i), e), \text{Release}) \rightarrow ([R], i + 1))\} \\ \lambda &= \{([R, r], i) \rightarrow \text{Resources}(r)\} \\ ta &= \{([R], i > 0) \rightarrow 0, \\ &\quad ([], i) \rightarrow \infty, \\ &\quad ([R], 0) \rightarrow \infty\} \end{aligned}$$

3. *Resource Handler*: The resource handler is responsible for keeping track of resources. Incoming requests are put in a queue, and, when resources are available, the request is processed. Processing a resource means that the resources are acquired in the

resource handler (i.e., a counter is decremented), and a reply is sent to the requester. When a request for a release is sent, the resource is immediately released (i.e., a counter is incremented).

The logic of the request handler can be complex, such as request prioritization, different resource types, and deadlock resolution. In our case, we only present a minimal example resource handler, managing a single type of resource, for which each request operates on exactly one resource instance. This Atomic DEVS model is given in Equation (5.10).

#### 5.5.4 Performance evaluation

After mapping the process model to the DEVS model and simulating it, the performance metric of transit time is obtained. Figure 5.37 shows the performance results of our running example, charted against the number of available resources. The chart shows the standard box plot features (i.e. minimum, first quartile, median, third quartile, maximum, and outliers) of the single simulation cases. For each scenario, 1000 simulations have been carried out.

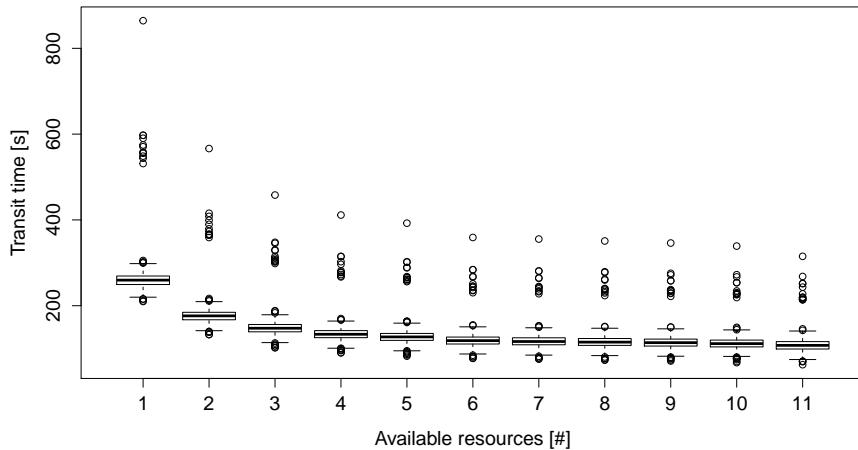


Figure 5.37: Results of the simulations.

The sharp decrease of the transit time is due to the more and more parallel threads spawned from the *HIL Simulation* activity (Figure 5.33). We spawn at most 11 concurrent activities (10 simulators and 1 model checker). This explains why with 6 available resources, time doesn't decrease further: we need at least two "phases" of executing the activities. Subsequently, when 11 resources are available, we see another minor decrease in time, since then all activities can then execute concurrently. More than 11 resources are not shown, as then we just have spare resources which are never used.

Note that there are several "bands" in which results are clustered. Indeed, the majority of the execution runs are clustered in the boxplots, which for the most likely scenario. However, we see that there are scenario's that cluster together: these values depend on the decision that was made at the end of the process. If we immediately decide that results

are fine, the lowest “band” of results is achieved. Similarly, the topmost results indicate that the decision was made to redo the modeling for several times. Of course, results vary depending on the model and used parameters.

Such results can be used in various ways for process optimization. Given an appropriate cost function of resources, one can identify the appropriate amount of resources to be added to the process for an optimal configuration. The chart shows no relevant decrease of the transit time after case 5, which (in the absence of a more complex cost function) may denote the optimum of the amount of added resources. The chart also shows a large variance in transit times, shown by the circles denoting the outliers. This is an indicator of brittle processes with a hidden hazard factor, which should be investigated.

### 5.5.5 Conclusions

In this paper, we presented an approach for simulating performance metrics of engineering process models. Our approach relies on the automated translation of engineering process models to DEVS models, and the capability of the latter to simulate stochastic performance metrics, such as transit time. Our approach supports each of the *van der Aalst* control patterns [187]. This means that our approach is compatible with the vast majority of process and process modeling formalisms. Our proof-of-concept implementation integrates with the FTG+PM framework seamlessly. We presented how stochastic behavior and resource constraints can be modeled in order to gain more realistic performance metrics.

In future work, we plan to further investigate DEVS simulations for different performance metrics, and feeding back the results to improve the process. The approach presented in this paper integrates with our inconsistency management framework [41], where inconsistencies and their quantitative metrics [42] serve as performance metrics to the process.

## 5.6 A multi-paradigm approach for modeling service interactions for engineering processes

In this section, we present **Contribution 8** of this thesis: a formalism for modeling complex engineering processes. The majority of this section has been published in [195].

Engineering processes aim at depicting how the various domain-specific models are used during development. Models are passed around in the process and are being worked on within the activities of the process. These activities are either manual or automated, and typically make use of various services offered by engineering tools. If modeled in an appropriate formalism, the process can be analysed and subsequently enacted [143]. The enacted process orchestrates the engineering services, thus enabling a higher level of automation in the flow of the modeling work in general.

Orchestration requires a detailed specification of the interaction protocol with external services. In manual activities, user input is required, often through a (visual) modeling and simulation tool (for example, to create a model). In automated activities, a service (or multiple services) might be invoked and communicated with in an automated way (for example, to run a simulation). Such interaction protocols exhibit timed, reactive,

and concurrent behaviour, making their formal analysis paramount in industrial-scale engineering processes. The analysis of the interaction protocols can improve its overall process with regards to transit time, scheduling, resource utilization, and overall model consistency. These interactions are, however, typically specified in scripts or program code, which interface with the API of the tools providing the services. Such an encoding of the interaction protocols inhibits their formal analysis.

In this section, we propose to explicitly model the external service interaction protocols in the activities of engineering processes using SCCD [196], a variant of Statecharts [85]. SCCD is appropriate for modeling timed, reactive, autonomous, and dynamic-structure behaviour, as it has native constructs available for it. This facilitates the implementation of the interaction protocols, and enables future analysis of the service orchestration. Additionally, we provide execution semantics for the overall process, expressed as an FTG+PMmodel, by mapping it to an SCCD model, augmented with process semantics. This avoids the need to define operational semantics for activity diagrams, which is non-trivial.

### 5.6.1 The Statecharts + Class Diagrams (SCCD) formalism

*Statecharts* is a formalism for modeling timed, reactive, autonomous systems, and was introduced by Harel [85]. Its main abstractions are states that can be composed hierarchically and orthogonally; transitions between these states that are either spontaneous, or triggered by an external event (coming from the environment), an internal event raised by an orthogonal component, or a timeout; and actions that are executed when a transition is executed.

While *Statecharts* is an appropriate formalism for describing the timed, reactive, autonomous behaviour of systems, it does not allow to model a system with dynamically changing structure. In many systems, objects are continuously created and destroyed. The SCCD formalism [196] extends the *Statecharts* formalism with the concepts of the *Class Diagrams* formalism (classes and relations, which model structure). Each class in the class diagram is associated with a definition of its behaviour (in the form of a *Statecharts* model). At runtime, an object can request for a class to be instantiated as an object, and relationships between classes to be instantiated as links between objects. Links serve as communication channels, over which objects can send and receive events. There is exactly one default class, of which an instance is created when the system is started by the runtime.

### 5.6.2 Motivating example

Our motivating example is the optimization of the number of traffic signals in a railway system. The system consists of sequences of railway segments, each guarded by a single traffic signal. For safety reasons, only one train is allowed per railway segment. Adding traffic signals increases the throughput of the system, though increases the cost of maintenance. The ideal number of traffic signals is therefore dependent on the characteristics of the system (*e.g.*, train inter-arrival time, acceleration, total length of the track).

The optimization is done by modeling the system with the DEVS formalism [216]. Our problem requires several atomic DEVSmodels, such as a generator, collector, railway segment, and a traffic signal, and a single coupled DEVSmodel, coupling these atomic

models together. This model is subsequently simulated for a fixed set of parameters, while varying the number of traffic signals over the total length. All simulation results are collected, the cost function is evaluated for all of them, and the number of traffic signals with the minimal cost is returned.

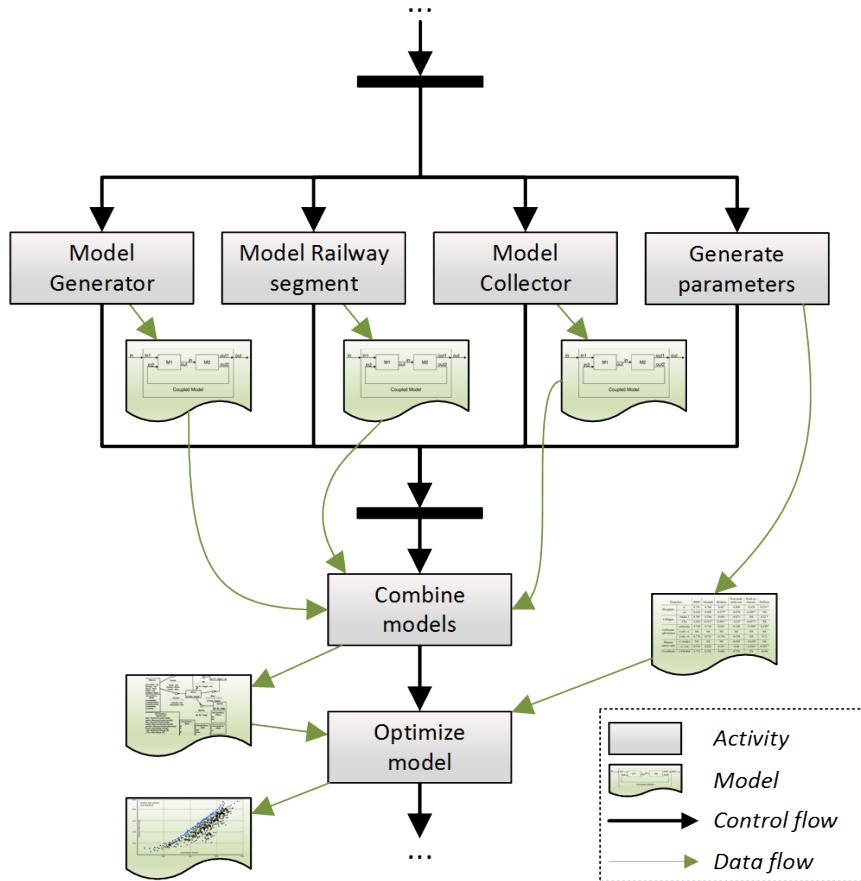


Figure 5.38: The process model of the example.

This process is shown in Figure 5.38, where we first design the various atomic models manually, though concurrently. As such, multiple engineers can model different aspects of the system concurrently. Additionally, a set of parameters is chosen, for which to simulate the model. Afterwards, the created atomic DEVSmodels are used to create the coupled DEVSmodel. It is this collection of models that is passed on to the optimization step, which plots out the costs of various configurations.

We implemented this example in the Modelverse [200, 197], our prototype Multi-Paradigm modeling tool. Simulations were performed using PythonPDEVS [198] as an external service.

### 5.6.3 Modeling activities using SCCD

We first turn to the definition of an activity. Activities are the atomic actions being executed throughout the enactment of the process. Up to now, we were agnostic of what is the content of the activity, as we merely require it to be executable. Most often, it is hard-coded in some programming language or provided as an executable binary. When control is passed to a specific activity, the activity executes.

#### 5.6.3.1 Problem statement

Activities can be hard-coded, but code is arguably not the optimal formalism to describe an activity. While activities can be limited to executing some local computation, it frequently requires external tool interaction. Such external tools can be anything, for example a (highly-optimized) simulator, or a modeling tool. In such cases, hard-coding the potentially complex interaction protocol is far from ideal. Indeed, the behaviour of protocols exhibits timing (*e.g.*, network timeouts, delays), reactivity (*e.g.*, responding to an incoming message), and concurrency (*e.g.*, orchestrating multiple tools concurrently).

In our running example, we see this exact problem occurring in the “*optimize model*” activity. In this activity, we want to optimize the cost for a given set of parameters, varying a single parameter within a given range. Concretely, we want to vary the number of traffic lights in the simulation, while keeping all other parameters fixed. In the end, the activity needs to return the optimal solution; that is, it returns the optimal number of traffic lights for the given set of parameters. In essence, the same simulation is ran with slightly different parameters. This is, however, embarrassingly parallel: each simulation run is independent of every other simulation run. Therefore, we desire to run some simulations in parallel. Doing this the usual way (*i.e.*, with code) is non-trivial: concurrency requires threads (which is problematic [115]), reactivity requires the use of a main loop (possibly with polling), and timeouts require interruptable sleep calls.

#### 5.6.3.2 Approach

The previous discussion illustrated that code is not ideal to specify timed, reactive, and concurrent activities, which is the case with service orchestration. We propose to use a formalism equipped with better support for these requirements: SCCD. Some activities are therefore ideally modeled with SCCD, where they can automatically make use of its features. On the implementation side, SCCD manages all concurrency, timing, and reactivity natively. Indeed, concurrency is supported by orthogonal components and dynamic structure, reactivity is supported by event-based transitions, and timeouts are supported by *after* events.

In our running example, we see that these features of SCCD are all required in the “*optimize model*” activity. Concurrency is required to spawn several instances of the simulator concurrently, and the number is only known at runtime, as it depends on the number of possible configurations. Reactivity is required to handle the results of these individual simulators, which should be aggregated. Timeouts are required to handle network timeouts and potential infinite simulations.

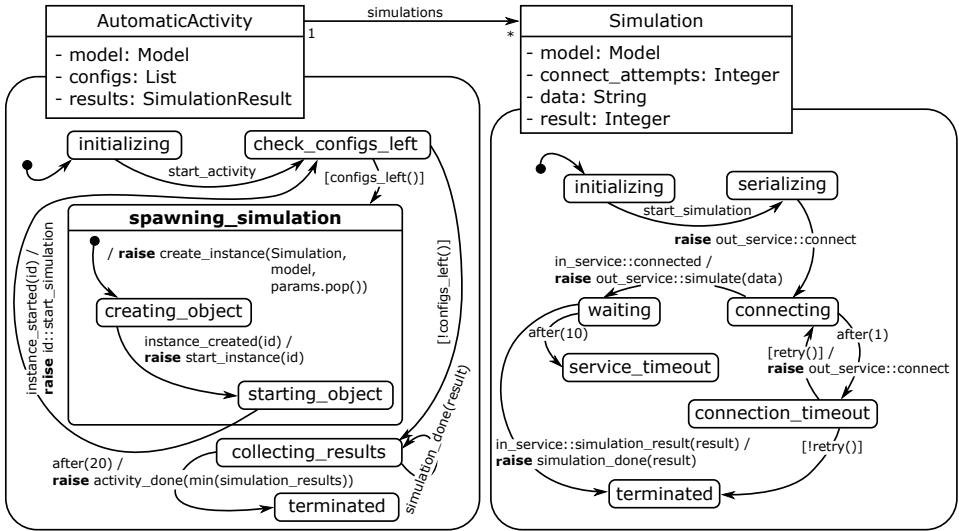


Figure 5.39: Automated activity: protocol implemented to communicate with an external service.

In Figure 5.39, we show how the example activity is modeled with SCCD. Thanks to SCCD, we can spawn an arbitrary number of “*Simulation*” objects, by sending out an event to the object manager, thereby allowing for dynamic structure (implementing *concurrent*). After a simulation is spawned dynamically, for each configuration to evaluate, we wait for results to come in, encoded in events (implementing *reactive behaviour*). Each of the spawned simulations serializes the model, and sends it to the actual external simulator, after which the simulator is started externally. If no response is received from the simulator during initialization before a timeout occurs, we retry the connection (implementing *timing*). If the simulation was started successfully, but no result comes in before a timeout occurs, we determine that the simulation has crashed, is stuck in an infinite loop, or ran out of memory. Independent of the reason, we determine that the simulation result is not the optimum, and subsequently ignore the simulation run. When all simulation results are in, or we have waited sufficiently long, we return the optimal parameter that we found.

#### 5.6.4 Mapping Processes to SCCD

Orthogonal to the previous section, where we modeled the contents of the activities using SCCD, we now look at the process model itself. The process model chains the different activities, dictating the order in which they should be executed, possibly concurrently. Of specific interest is the fork/join operation, which executes multiple activities concurrently and synchronizes when both have finished. This is ideal for manual activities, for which multiple developers might be involved, who can now model concurrently.

### 5.6.4.1 Problem statement

Despite the advantages of concurrent manual activities, implementing this in a truly parallel fashion is non-trivial. Basic implementations merely dictate an arbitrary order between different concurrent activities, without actually executing them in parallel. This was originally the case in our prototype tool, the Modelverse, because true concurrency is difficult and relies on many platform characteristics. Examples are the choice between processes or threads, their interleavings, how the implementation platform supports parallelism, and how data is shared between activities. These are only a small selection of crucial questions regarding the implementation of process enactment. A significant investment to implement and maintain this infrastructure is needed if processes are implemented using traditional (code-based) techniques.

For our running example, this is shown in the concurrent manual activities in the beginning of the process: creating the various DEVSmodels. These models are independent, and can easily be created in parallel. Nonetheless, if there is no support for activities to run concurrently, all work is effectively sequentialized, significantly increasing the duration of the overall process.

### 5.6.4.2 Approach

The problem arises due to the lacking native support for concurrency in many implementation languages. As such, implementing process model enactment requires many workarounds to achieve true parallelism. We note, however, that languages do exist that natively support notions of concurrency, for example SCCD. Nonetheless, SCCD was not designed to model processes, and is therefore not suited for direct modeling. In summary, we want users to model using activity diagrams, as they are used to, but for execution purposes, we transform the modeled process to an SCCD model. This transformation defines denotational semantics for process models, instead of operational semantics (an executor).

While other such languages exist, we favour SCCD as this allows us to reuse the SCCD execution engine, needed to execute activities. Additionally, we see many future opportunities for our approach if both orthogonal dimensions (modeling activities with SCCD, and mapping the process to SCCD) are combined: both share the same (hierarchical) formalism, and can therefore potentially be flattened.

Mapping activity diagrams to SCCD can be achieved through the use of model transformations, which are often referred to as the heart and soul of MDE [165]. With model transformations, a Left-Hand Side (LHS) is searched throughout the model, and, when matched, the match is replaced with a Right-Hand Side (RHS), if the Negative Application Condition (NAC) does not match at the same time. In our case, the LHS consists of activity diagrams elements, such as the *activity* construct, while the RHS copies the activity diagram construct (thereby leaving the activity diagram intact) and creates an equivalent SCCD construct (*i.e.*, an orthogonal component). Defining such a mapping is significantly less work than defining operational semantics from scratch, as we will show. Additionally, by mapping to SCCD, there is only one implementation of an executor for timed, reactive, autonomous, dynamic-structure behaviour that must be maintained (the SCCD executor).

A naive mapping to SCCD would map forked activities to orthogonal components, each spawning and managing the execution of the activities; joins synchronize the execution by transitioning from the end states of these components. While intuitive, this mapping runs into problems with non-trivial concurrent regions. For example, consider two parallel forks that interleave: the two forks cannot be independently mapped, as they interact with one another, resulting in a different mapping to orthogonal regions.

We define a more generic mapping, based on orthogonal components, one for each activity diagrams construct. The order in which the orthogonal components are enabled, is defined by the condition that is present in the orthogonal component itself. Each orthogonal component checks whether it has the “execution token”, and if so, it passes on the token. All orthogonal components execute concurrently, meaning that if multiple tokens exist, multiple orthogonal components operate concurrently. Depending on the type of construct, the behaviour changes: activities execute and pass on the token upon completion, a fork splits the token, a join merges tokens, and a decision passes the token conditionally. In the remainder of this section, we describe our transformation rules for each activity diagram construct in detail.

#### 5.6.4.3 Transformation rules

The following transformation rules are executed in the presented order. Before we actually start the translation, however, we first perform a minor optimization step: subsequent fork operations are merged into a single fork. This is not performed for performance considerations, but makes the mapping slightly easier. This optimization thereby removes subsequent forks, allowing us to skip this case in the remainder of the mapping. The same happens for join nodes.

#### Optimization

Figure 5.40 presents the optimization of fork nodes. The first (topmost) rule makes sure that the first fork directly links to all targets of the second fork, removing the target from the second fork. This rule keeps the model semantically equivalent, as the second fork now has no successors. In the second (bottommost) rule, an empty fork node is removed, as it has no outgoing edges any more. This rule again maintains semantic equivalence, as the second fork has no successors left. Similar rules exist for the optimization of fork nodes.

#### Orchestrator

Figure 5.41 presents the transformation rule for the orchestrator, which executes once. Each subsequent transformation rule extends a single composite state with an orthogonal region. The orthogonal regions execute all elements of the activity diagram in parallel, waiting for a condition to become true. The first step consists of creating the composite state and providing it with an orthogonal region that catches a spawn event, and performs the spawning of an activity. By defining this code here, it does not have to be reproduced throughout the other orthogonal regions, maximizing reuse.

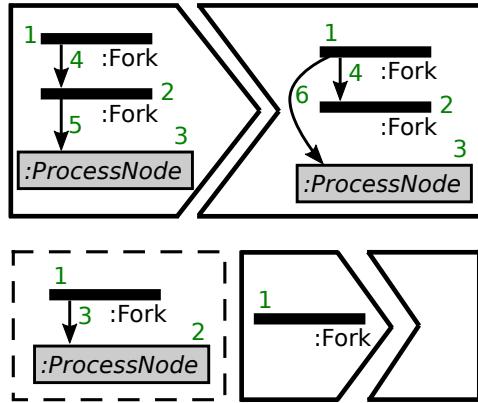


Figure 5.40: Optimize rules.

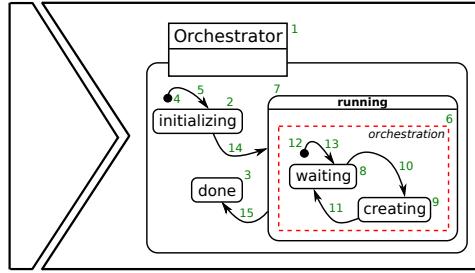


Figure 5.41: Orchestrator rule.

## Activity

Figure 5.42 presents the transformation rule that executes for each activity. Activities are relatively easy to map, as they merely require the spawning of their associated activity (which, in our case, is modeled by another SCCD class). This is achieved by sending a spawn event to the orchestrator, and transitioning to a “running” state. We stay in this state until we have determined that the spawned activity has terminated, after which we mark the current activity as executed (*i.e.*, we pass on the token).

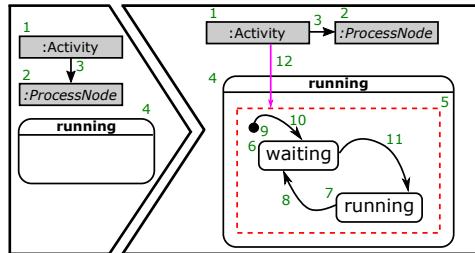


Figure 5.42: Activity rule.

### Fork

Figure 5.43 presents the transformation rule that executes for each fork node. Forking requires a single token to be distributed among all of its successors, without doing any computation itself. As such, our transformation rule adds an orthogonal component which continuously polls whether or not it has received the token. If it receives the token, it immediately passes the token to all of its successors simultaneously.

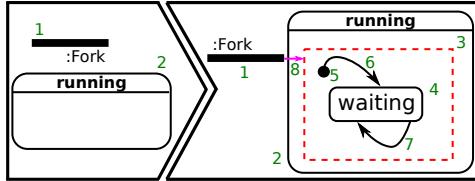


Figure 5.43: Fork rule.

### Join

Figure 5.44 presents the transformation rule that executes for each join node. Joining is slightly more complex: it has to check for multiple tokens, before becoming enabled. When enabled, it consumes all of these tokens and passes on the token to its own successor, of which there is only one.

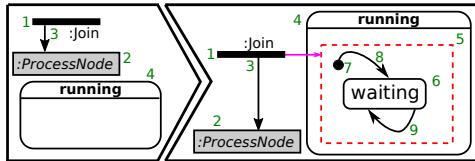


Figure 5.44: Join rule.

### Decision

Figure 5.45 presents the transformation rule that executes for each decision node. The final construct that we have to map, is the decision node. Similar to all previous nodes, we check whether we have a token to start execution. Depending on the input data that we receive, we decide to pass on the token to either the *true*- or the *false*-branch.

#### 5.6.4.4 Results

The resulting mapped model for our motivating example is shown in Figure 5.46. The Orchestrator class is the default class of the SCCD and spawns the other classes (i.e., the activities) depending on its orthogonal components. These other classes are, for example, the optimization activity previously described in Figure 5.39. Other classes are not shown due to space restrictions.

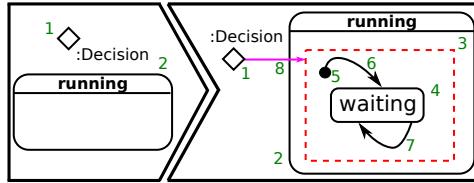


Figure 5.45: Decision rule.

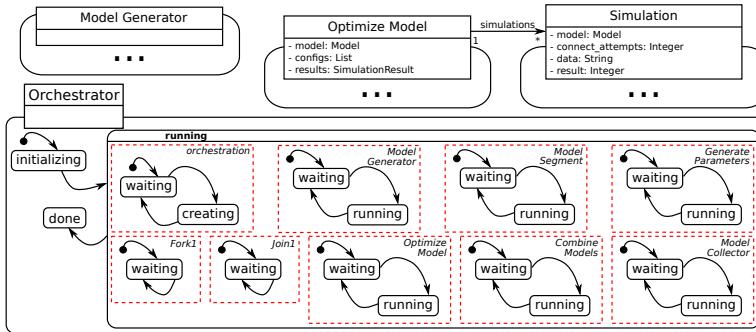


Figure 5.46: Process model from Figure 5.38 mapped to SCCD.

## 5.6.5 Conclusions

In the context of MPM, service orchestration is essential for the combination of multiple external tools. Nonetheless, current approaches do not sufficiently address the challenges it poses: timed, reactive, and concurrent behaviour. In this paper, we propose an approach for handling these problems by two contributions, based on SCCD (a *Statecharts* variant), which has native notions of timing, reactivity, concurrency, and dynamic structure. First, activities themselves are modeled using SCCD, allowing external service protocols to be more effectively specified. Second, the process model is transformed into an equivalent SCCD model for execution. This preserves the modeling abstractions provided by activity diagrams, while gaining the execution of SCCD.

In future work, we plan to consider the benefits of combining these two orthogonal dimensions of our approach. Indeed, as both the process and activities are modeled in SCCD, they can be combined into a single SCCD model. This single SCCD model can subsequently be analysed [147] or debugged [135], without any additional work. To achieve the valid and sound construction of this combined SCCD interaction/process model, composition rules of the single interaction SCCD model need to be investigated. Our previous work on process-oriented inconsistency management in MPM settings [41] is a prime candidate to be augmented with such an approach. *Software Process Improvement* (SPI) techniques in general can greatly benefit from our approach as well.

## 5.7 Inconsistency tolerance by quantified semantic inconsistency metrics

The majority of this section has been published in [45].

Finkelstein [70] suggests that instead of simply removing inconsistencies from the system from time to time, one should manage consistency. This entails characterization and detection of inconsistencies before resolving them, and the subsequent analysis. Inconsistencies are, however, stateful entities that might occur, evolve and later potentially disappear as the natural consequence of a design workflow. This gives room for temporarily *tolerating* them [12], i.e. allowing inconsistencies to exist for a period of time, which promises lower resolution costs by (i) postponing resolution to a more appropriate phase of the design process; or, in some cases, even (ii) completely avoiding resolution when specific inconsistencies get resolved on their own.

Deciding whether or not to intervene in an inconsistent situation requires information about (i) how severe the inconsistency of the whole model space is; and (ii) what are the chances that the inconsistency gets resolved without intervention. In this paper, we focus on the first question and present a quantitative approach for assessing the severity of semantic inconsistencies. For that, we formalize the notion of (in)consistency in multi-view modeling by defining a relation between a property of the whole system, the satisfaction of the property by individual views, and the consistency between views. We identify characteristic behavioral patterns of properties in terms of their consistency and use these patterns for guiding the tolerance. The results of this paper serve as formal foundations to implement a consistency management framework for collaborative modeling scenarios with multi-paradigm and multi-abstraction characteristics.

### 5.7.1 Motivating examples

#### Collaboration models

Our example concerns the development of complex and heterogenous systems such as CPS involving the collaboration of several development teams. This collaboration can be supported by cloud-based model-based development environments such as the one presented in [38]. Such environments allow the definition of viewpoints specifying how different views of the system under development can be constructed to be used by the various teams to interact with the models of the system. These environments can typically support two different working modes. For the *simultaneous* mode, any update performed on a view is immediately sent to the server for being applied to the model and changes are reflected on all other depending views. This is the way well known collaborative tools work, such as Google Docs or Overleaf, which were used to write this paper collaboratively.

However this simultaneous mode may not meet the needs of all development organizations. For example, large and complex systems are often decomposed into subsystems developed by different entities (e.g., manufacturers, suppliers, subcontractors, etc.). Furthermore, due to intellectual property protection concerns, the models may never be allowed to cross the boundaries of an enterprise local network. Then, a system integrator needs to plan ahead of time a series of virtual integration steps where the models developed by all external entities

are integrated and analyzed for various properties. Consistency of the system as a whole then needs to be checked as the models have evolved independently for a given period of time. According to the analysis results, decisions and updates will then be made to restore consistency. We call this mode *delayed commit*. The frequency of such virtual integration steps may vary but has to be carefully considered, e.g., to avoid the large costs potentially induced by the rework of highly inconsistent models. In this mode, providing a measure that quantitatively determines how inconsistent the models are is highly relevant.

### The Automated Guided Vehicle

Our running motivational example is the collaborative model-based development of an Automated Guided Vehicle (AGV) (Figure 5.47). According to the stakeholder requirements specification, the AGV needs to be able to transport a number of objects by following a given trajectory with a given maximal tracking error. Furthermore, it needs to have a given autonomy as defined by the number of trajectories that can be followed before needing to recharge the AGV's empowering component. Finally, the mass of the AGV must not exceed a given value because, for example, it will be too heavy for the setting in which it is planned to operate.

To meet these requirements, an initial design is provided consisting of a custom mechanical frame (Figure 5.47), an off-the-shelf electrical battery, an off-the-shelf electrical motor, a drive train and a drive train control system. A set of viewpoints for this design is then defined in the collaborative environment to be used for design activities such as requirements specification, electric motor selection, battery selection, mechanical frame design, drive train design and drive train controller design. These are listed in the header row of Table 5.1.

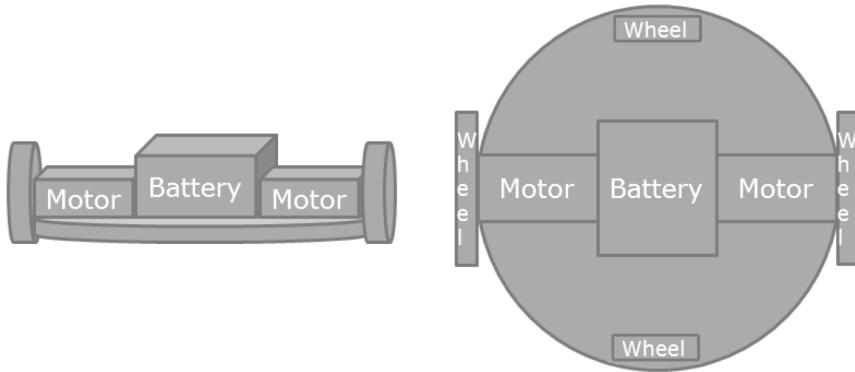


Figure 5.47: Overview of the Automated Guided Vehicle example.

#### 5.7.2 System properties and consistency statements

Herzig et al [90] define inconsistency as two or more statements that are not jointly satisfiable. In our approach, we provide such set of statements per system *property* and concerned *viewpoint* as defined in the cells of Table 5.1 for the AGV example. Overall

	Requirements ( $V_R$ )	Mechanical Frame ( $V_{MF}$ )	Battery ( $V_B$ )	Motor ( $V_M$ )	Drivetrain ( $V_{DT}$ )	Drivetrain Control ( $V_{DTC}$ )	Integrated Vehicle ( $V_{IV}$ )
<b>Battery Support Size</b>		= Battery Support Size	$\geq$ Battery Size				
<b>Electrical Current Capacity</b>			$\leq$ Battery Current Capacity	$\geq$ Minimum Current Capacity Required by Motor			
<b>Power</b>				$\leq$ Motor Power		$\geq$ Maximum Motor Power Required by Drive Train Control System (e.g. obtained from simulation)	
<b>Mass</b>	$\leq$ Maximum Vehicle Mass (obtained from a Requirement)	$>$ Frame Mass	$>$ Battery Mass	$>$ Motor Mass	$>$ Drive Train Mass	$\leq$ Maximum Vehicle Mass Supported by Drive Train Control System (e.g. obtained from simulation)	= Vehicle Mass (computed from the sum of its constituents)

Table 5.1: Example consistency properties and statements about their values for each relevant viewpoint.

consistency then consists in ensuring consistency for each given property of interest of the system. For a given property, consistency can be checked for each pair of viewpoints providing a *statement* about the property. From the two statements of the viewpoints, a compound statement can be derived about the consistency of both viewpoints.

For example, consider property  $P = \text{BatterySupportSize}$  for the AGV (first row of Table 5.1). Let us assume that there is a single underlying model for the mechanical frame viewpoint and a separate model for the battery viewpoint. The battery support size provided by the mechanical frame view is then simply the value of the corresponding property in the underlying mechanical design model. On the battery view however, a less precise information is provided as all that can be stated for a consistent design is that it should be greater or equal to the size of the battery. Therefore, only a range of values can be inferred from the battery viewpoint. Composing these two statements, we derive the compound statement

$$\text{BatterySupportSize} \geq \text{BatterySize},$$

which can be evaluated at design time from the involved views  $V_{MF}$  and  $V_B$  to evaluate consistency. In other words, a value from the battery view outside the range provided by the mechanical frame view will indicate an inconsistent system.

Generalizing this example, we can say that for a given property  $P$  of the system under design, it is possible to automatically derive for each pair of viewpoints providing a statement

about  $P$  a compound statement about the consistency of both views with respect to  $P$ . This compound statement will involve all properties involved in the composed statements. For example, for  $P = \text{BatterySupportSize}$ , we obtain a consistency statement about the pair of properties  $\text{BatterySupportSize}$  and  $\text{BatterySize}$ . In Section 5.7.4, we further show how such statement can be adapted to provide a quantification of inconsistency.

It should be noted that such considerations are independent from the source of inconsistency. For example, it can originate from the actual values of the underlying models used by the views, but also simply from the fact that the views are outdated due to network latency or delayed commit between the views in the cloud-based development environment. In order to explore different types of inconsistency tolerance patterns as proposed in Section 5.7.5, we will therefore consider that the two different modes (simultaneous and delayed commit) for the cloud-based multi-view modeling environment are used for two different development processes. Example use cases for this are provided in Section 5.7.5.

### 5.7.3 Other applicable cases

While our motivating example concerns collaborative development, we emphasize here that what we provide is applicable to many other cases, thus making this proposal even more relevant. One of them is that of *runtime* models and views such as those employed in computer games implying a variety of consistency concerns. These are most apparent in network-based, multiplayer games, which are both realtime environments and distributed systems, and as such bring a number of consistency concerns due to network latency, lossy communication protocols, and bandwidth limitations. Even single player games induce consistency issues, with modern designs organized through a separation of domains in order to provide good, domain-specific performance and behaviour, a distinction quite visually evident in, for example, the way collision domains are simplified abstractions of the representation domain [66]. Games make a further interesting and motivational subject due to the way consistency is tolerated - the perception-oriented, entertainment focus of game software means subtle inconsistency is not necessarily a problem, and weaker or heuristic consistency guarantees are often viewed as an acceptable trade-off for increased performance or broader scope in implementation.

### 5.7.4 A quantitative model for assessing inconsistencies

Based on our running example, we can now provide a model for quantifying inconsistency. This can be achieved as explained previously for a given system property such as those listed in Table 5.1. We first consider consistency for a given system property  $P$  and two given views  $V_1$  and  $V_2$  each providing a statement about  $P$ . The consistency statement composed from the statements of  $V_1$  and  $V_2$  about  $P$  then provides a set of two properties that must be evaluated to evaluate the consistency statement. We then propose a metric combining these two properties to quantify the degree of inconsistency. Such measure can later be used for assessing the importance of the inconsistency and support its management.

#### 5.7.4.1 Formal definitions

The definition of consistency is based on a *distance function* that takes the two views and provides a measure value. In this simple setting we need a simple domain for measures that can be used in a compositional and associative way.

**Definition 5.** (*Measure*) A given set of values  $\mathbb{M}$  with operations  $0 : \rightarrow \mathbb{M}$ ,  $+ : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}$  ( $0$  neutral,  $+$  associative and commutative) and an order relation  $\leq$  on  $\mathbb{M}$  is called a measure.

**Definition 6.** (*State distance*) Given two state domains  $\Sigma_1$  and  $\Sigma_2$ , a distance is a function  $\delta : \Sigma_1 \times \Sigma_2 \mapsto \mathbb{M}$ .

From the state distance we can define by extension distances on finite prefix of traces. By trace, we mean a sequence of performance values [203] a semantic property exhibits as the underlying model changes. This distance is a natural extension that can express the observation on a trace with a finite window starting from the beginning of the trace. The size of the window  $\lambda$  is a parameter of such distance.

**Definition 7.** (*Sequence of ordered points*) A sequence of ordered points (or index on a trace) is a word  $I = i_1, i_2, \dots, i_n, \dots \in \mathbb{N}$  s.t.  $\forall k, i_k < i_{k+1}$ .

**Definition 8.** (*Window*) Given a sequence of observations  $O = o_1, o_2, \dots, o_n$ , a window  $w$  of length  $\lambda$  is defined as a prefix of length  $\lambda$  of  $O$ .

**Definition 9.** (*Trace distance*) Given two set of traces  $\Sigma_1^\omega$  and  $\Sigma_2^\omega$  generated by two properties  $\rho_1$  and  $\rho_2$ , respectively, a distance on trace for a given window of lenght  $\lambda$  is a function  $\delta_\lambda : \Sigma_1^\omega \times \Sigma_2^\omega \mapsto \mathbb{M}$  defined as:

$$\delta_\lambda(\rho_1, \rho_2) = \sum_{i=0}^{\lambda-1} \delta(\rho_1(i), \rho_2(i)),$$

where  $\rho(i)$  denotes the  $i$ th observation of  $\rho$ .

The above definition can be extended to continuous systems as follows:

$$\delta_\lambda(\rho_1, \rho_2) = \int_{i=0}^{\lambda} \delta(\rho_1(i), \rho_2(i)) d_i.$$

**Definition 10.** (*Consistency*) Two views  $V_1$  and  $V_2$  are said to be consistent iff

$$\forall \rho \in [\![M]\!]_{V_1 \times V_2} : \delta_\lambda(\rho|_{V_1}, \rho|_{V_2}) = 0.$$

That is,  $V_1$  and  $V_2$  are consistent iff every property  $\rho$  has the same value in both views.

#### 5.7.4.2 Measuring distances

Inconsistency management requires providing a meaningful consistency measure given the two properties and the states for evaluating them. Such measure should take into account

the amount of development work required to fix the inconsistency. This depends on several factors such as which activities must be performed to fix the design, which in turn depends on which part(s) of the design will be changed to resolve the inconsistency (e.g. for our example battery selection or battery support from mechanical design or both). One way to evaluate this is to take into account the cost of the involved development activities that will need to be performed. This set of activities can be determined by the graph of dependencies of the involved properties and their dependency to other properties. However, this complex task is left for future work. For now, we only provide illustrative simplified examples showing how simple measures can be derived for the different properties and viewpoints of Table 5.1.

For our AGV example, the aforementioned state trace is defined by the successive model states corresponding to the versions of the system design evolving through the activities performed along a development time axis.

**Battery support size** For the battery support size property, let us assume that the sections of the battery and its support are of circular shape and can therefore be characterized by a single numerical value of the section diameter. We compose the statements of viewpoints  $V_{MF}$  and  $V_B$  of Table 5.1 to obtain a Boolean consistency statement about the pair of properties  $BatterySupportSize$  and  $BatterySize$  as:

$$\rho_{BatterySize} \leq \rho_{BatterySupportSize}$$

where  $\rho_{BatterySupportSize}$  and  $\rho_{BatterySize}$  are the property values obtained from  $V_{MF}$  and  $V_B$  respectively. In such case, a simple measure of inconsistency can be given assuming that the amount of rework required for changing the battery support size of the mechanical design to account for the battery size is proportional to the difference in the diameters of the battery and battery support. Thus, the consistency statement can be transformed to provide a metric for the amount of inconsistency as the difference between the battery and battery support diameters:

$$\delta = |\rho_{BatterySize} - \rho_{BatterySupportSize}|$$

**Vehicle mass** For the AGV mass property, we first consider the statements from the viewpoint of the drive train control system  $V_{DTC}$  and the integrated vehicle mechanical design  $V_{IV}$ . In a similar process as for the battery support size we derive the consistency statement:

$$\rho_{VehicleMass} \leq \rho_{MaxVehicleMassDriveTrainControl}$$

And with similar simplification as for the battery support size property that the amount of rework to account for fixing an inconsistent design is proportional to the difference between the masses, we derive the measure:

$$\delta = |\rho_{VehicleMass} - \rho_{MaxVehicleMassDriveTrainControl}|$$

### 5.7.5 Towards tolerance of inconsistencies

In this section we discuss how the quantitative model in Section 5.7.4 can be used to tolerate semantic inconsistencies. We support the notion of two types of inconsistency

tolerance.

In *parameter tolerance*, the definition of inconsistency is weakened to allow slight deviations from desired values of parameters, i.e. performance values of semantic properties [203]. In *temporal tolerance* scenarios, semantic inconsistencies are allowed for a certain amount of time before intervening and resolving inconsistencies, making use of the fact that inconsistencies might occur, evolve and later potentially disappear as the natural consequence of a design workflow.

### 5.7.5.1 Tolerating parameter inconsistencies

The idea of parameter tolerance embraces the presence of slight deviations from desired values of properties while still considering the situation a consistent one. In our specific case this means the distance between two views (properties) is not exactly 0, but stays within a parameter  $\beta$ .

Parameter tolerance enables reasoning about *how much* deviation is to be tolerated. For this purpose, Definition 10 can be weakened as follows.

**Definition 11.** (*Bounded consistency*)  $V_1$  and  $V_2$  are  $\beta$ -bounded consistent for  $M$  iff

$$\forall \rho \in \llbracket M \rrbracket_{V_1 \times V_2} : \delta_\lambda(\rho|_{V_1}, \rho|_{V_2}) \leq \beta,$$

where  $\beta$  is a measure. (See Definition 5.) The above definition allows the distance of two properties being within  $\beta$ , instead of restricting it to 0.

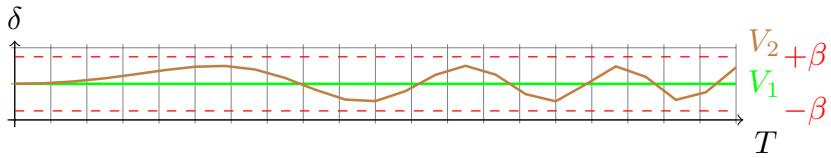


Figure 5.48:  $\beta$ -bounded consistency of two views.

From our running example, this occurs for the mass property when a high level of abstraction model of the drive train control system is used that, when executed, does not meet real-time deadlines for the given inertia due to the mass of the AGV. While the high level of abstraction model is functionally correct, it does not react fast enough due to poor execution time. The inconsistency with the mass obtained from the integrated vehicle view as evaluated according to the equation of Section 5.7.4.2 giving the measure of inconsistency may be tolerated at this level of abstraction up to a given bound  $\beta$ , otherwise it is well known by an experienced designer that no implemented system can ever execute fast enough for the given AGV inertia.

Such parameter inconsistency tolerance is a frequently encountered technique in engineering practice, albeit without explicitly reasoning about it. It is the tacit domain knowledge of stakeholders (engineers) that allows tolerating parameter inconsistencies. The caveat is, however, that without proper formalization, the automation of tolerance mechanisms is not possible.

Contract-Based Design (CBD) [17, 51] can be seen as an enabling methodology to formalize an agreement upon the allowed parameter tolerances. By definition, an agreement consists of a set of assumptions and guarantees, called a contract, defining under which conditions a system promises to operate satisfying desired parameters. Typically, a contract and its content, e.g. the boundaries of certain parameters, is defined prior to the design phase. When the relation between parameters is known, e.g. using an ontology [203], one can reason about the tolerated (in)consistency of parameters during design.

### 5.7.5.2 Temporal tolerance of inconsistencies

When inconsistencies emerge it may be also desirable to temporarily tolerate them, even if the extent of inconsistencies (i.e. the distance between views) exceeds the boundary  $\beta$  as defined previously. Temporal tolerance enables reasoning about *how long* a deviation is to be tolerated. For this purpose, we investigate characteristic behavioral patterns of inconsistencies in the temporal dimension. The patterns can be used in a *prescriptive* manner, i.e. specifying the expected behavior of properties and consider intervention when the expected behavior is not followed. In each pattern, we assume  $\beta$ -bounded consistency. (See Definition 11.)

**Exact consistency** The most simplistic behavioral pattern of properties wrt their consistency is when the consistency is maintained constantly.

**Definition 12.** (*Exact consistency*)  $V_1$  and  $V_2$  are exact consistent for  $M$  iff

$$\forall \rho \in \llbracket M \rrbracket_{V_1 \times V_2}, \forall t \in T : \delta_\lambda^t(\rho|_{V_1}, \rho|_{V_2}) \leq \beta.$$

That is,  $V_1$  and  $V_2$  are exact consistent iff the consistency is preserved constantly, i.e. for every timestamp  $t \in T$ .

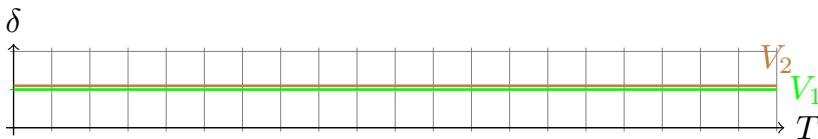


Figure 5.49: Exact consistency of two views with  $\beta = 0$ .

For our AGV example, this corresponds to the case where the cloud-based multi-view development environment is operating in simultaneous mode. In such case, the consistency of the properties defined in Table 5.1 is consistently monitored for each pair of views as the design of the system evolved and immediate resolving actions must be performed in case of any detected inconsistency.

In practice, it is however often not efficient to enforce exact consistency during system design, especially in complex multi-paradigm and multi-abstraction settings. The following patterns formalize cases more likely to be encountered.

**Eventual consistency** Eventual consistency guarantees that at some point in the future the two views become consistent.

**Definition 13.** (*Eventual consistency*)  $V_1$  and  $V_2$  are eventually consistent for  $M$  iff

$$\forall \rho \in \llbracket M \rrbracket_{V_1 \times V_2}, \exists t \in T : \delta_\lambda^t(\rho|_{V_1}, \rho|_{V_2}) \leq \beta.$$

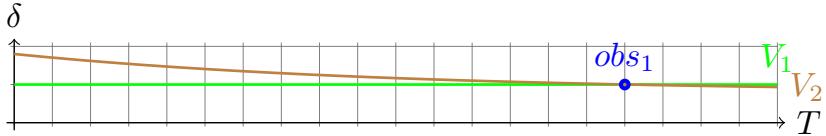


Figure 5.50: Eventual consistency of two views with  $\beta = 0$ .

For our AGV example, eventual consistency with  $\beta = 0$  will be required when the drive train controller design is refined into implementation code and the combined code and execution platform result in meeting the real-time deadlines for the total mass of the AGV.

Demanding eventual consistency is also a typical scenario when parallel branches of design are to be integrated at some point in the design process. This leads to the definition of the repetitive and regular consistency patterns presented below.

**Repetitive and Regular consistency** Similarly to the eventual case, consistency of views may form a repetitive pattern, i.e. views may be consistent at given points in time, but may deviate in between.

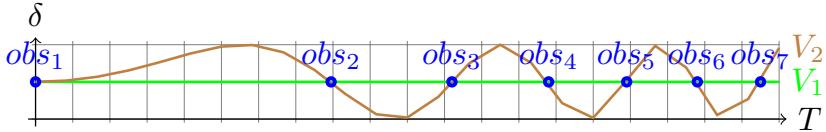


Figure 5.51: Repetitive consistency of two views.

**Definition 14.** (*Repetitive consistency*)  $V_1$  and  $V_2$  are repetitive consistent for  $M$  wrt a sequence of ordered points  $I$  (see Definition 7) iff

$$\forall \rho \in \llbracket M \rrbracket_{V_1 \times V_2}, \wedge_{i \in I} \delta_\lambda(\rho^i|_{V_1}, \rho^i|_{V_2}) \leq \beta.$$

For our AGV example, this corresponds to the case where the development of the different subsystems is subcontracted to external entities and virtual integration steps, at which consistency must be restored, are planned in advance in an aperiodic manner.

The repetitive pattern can be more regular w.r.t. to the number in the sequence, in this case it means that the index  $I$  is such that  $\forall i \in \mathbb{N}, I(i) = \tau * i$ .

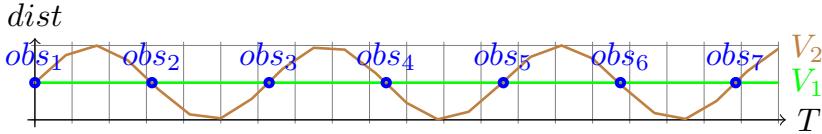


Figure 5.52: Regular consistency of two views.

For our AGV example, this corresponds to the same case as for repetitive consistency but with periodically planned virtual integration phases.

### 5.7.6 Discussion

The quantification approach of semantic inconsistencies described in this paper serves as foundations for a comprehensive inconsistency tolerance approach that can be adopted in state-of-the-art consistency management frameworks. Our work focuses on two types of inconsistency tolerance: parameter inconsistency tolerance and temporal tolerance in general.

Toleration methods for parameter inconsistencies can be directly implemented based on the theory described in Section 5.7.5.1. A requirement to such methods is the explicit notion of semantic traces, which can be achieved by explicit modeling of operational semantics, described e.g. in [47, 207].

In the following, we discuss the required extensions and future directions to the current results to make the foundations of temporal tolerance described in Section 5.7.5.2 applicable in complex collaborative modeling settings.

#### Explicitly modeled processes

Explicitly modeled processes enable various analyses required for reasoning about temporarily tolerating inconsistencies. First, they depict how models *evolve* throughout the development. The precedence relationships between various activities modifying the models are made clear, which helps understanding where and how inconsistencies occur exactly. Deciding whether to resolve an inconsistency or further tolerate it, eventually depends on the added value a fixed inconsistency carries. If the cost of resolving an inconsistency is high, it may be more appropriate not to intervene and let inconsistencies exist for a while. The efficiency of resolution depends on numerous factors, one of them being the *formalisms* (i.e. domain-specific languages) used throughout the process. Some formalisms may support more efficient resolution of a specific type of inconsistencies than others and therefore, the analysis of the formalisms used in specific design steps may help identifying when it is most efficient to address the issue of detected inconsistencies. Richer process models enable explicit modeling of this information [42], along with the notions of actors and resources of the process [9].

### Specification and evaluation of temporal tolerance rules

Specifying temporal tolerance rules makes only sense if those rules can be evaluated at runtime. Reasoning about temporal structures is traditionally addressed by various forms of temporal logic [77, 5, 158], which enable concise proofs and automated theorem proving with off-the-shelf software [96]. In nowadays' engineering practice, however, more high-level and domain-specific specification languages are desired. Such languages can be designed in accordance with the Dwyer specification patterns [53] for finite-state verification that occur commonly in the specification of concurrent and reactive systems. For evaluating temporal tolerance patterns, the techniques of trace matching and complex event processing can be used, as presented in our other work [44].

### Automation of tolerance specification

Tolerance rules applied to single properties usually influence tolerance rules of other, related properties. For example, in the running example, the "battery size" is related to the "battery mass" and inconsistencies between "size" could be also imply inconsistencies in "mass", which may not be tolerated in the same manner. Manual reasoning about the influence relationships of tolerance rules will result in scalability issues in real, industrial-sized models. Automation of specifying tolerance rules is, therefore, a critical requirement for an efficiently tolerating inconsistencies. Combinatorial optimization techniques and design-space exploration can aid the process of tolerance specification, and can provide a mechanism for exploring incompatible tolerance rules.

### Predictive impact analysis

The real challenge of answering whether or not to intervene at a specific point of the design process to resolve inconsistencies, is to understand how the inconsistencies can evolve in the subsequent steps of the process. Inconsistency resolution should be carried out only if the views are potential subjects of further diverging, but it should be postponed if the views can converge as a result of subsequent activities. With respect to the quantified distance metric, this raises the need for predictive techniques to be incorporated in our distance models to estimate how the impact of an inconsistency can evolve. In its most simplistic form, techniques of time series analysis can be employed [30]. In more complex scenarios, the propagation of inconsistencies may be required to be predicted too.

### Resolution scheduling

Another important cost-related factor is the resolution scheduling strategy of inconsistencies. Mens et al [127] investigated how a small number of inconsistencies can lead to hard resolution scheduling problems, including cyclic dependencies between inconsistencies, which make naïve resolution techniques fail. Consequently, the analysis of resolution plans is important to understand the feasibility of resolution and provides essential information on whether or not to execute resolution at the given time of the process.

### Proving global (in)consistency

Proving the consistent (or inconsistent) state of the models is a natural extension of the quantification model. By evaluating local consistency and defining the algebraic rules of *composition* of views [17], information about the global consistency can be inferred in an automated fashion. Apart from the composition rules of views, proving global (in)consistency depends on how the semantic domains of views overlap as discussed in [148].

### 5.7.7 Conclusions

In this section, we presented a quantification model for semantic inconsistencies in multi-view settings of collaborative system design. The model serves as the foundation of a more comprehensive inconsistency tolerance methodology. By elaborating on characteristic examples we concluded that the model is well-suited for supporting the notion of parameter inconsistency tolerance and also tolerating inconsistencies in the temporal aspect.

This section reported the results of a work in progress and therefore, the links to the state-of-the-art have been explored, as well as the further directions to be investigated. One particularly important direction concerns the measure of level of inconsistency. Our simplified example measures need to be extended to include a study on how to relate a measure of inconsistency with the cost of fixing it given an actual design, the other dependent consistency properties and their dependency graph, and given an inconsistency resolution plan expressed as a sequence of activities to be performed on the involved views. Adequate modeling of all these artifacts and corresponding analysis methods will be required to better support this task.

# Chapter 6

## Case study

In this section, we elaborate on the applying our methods on an industrial-scale engineering problem. There are two case studies discussed in this section, using the same problem, discussed in Section 6.2 and Section 6.3, respectively.

### 6.1 Introduction

We use a case study of the design of an automated guided vehicle (AGV), shown in Figure 6.1.

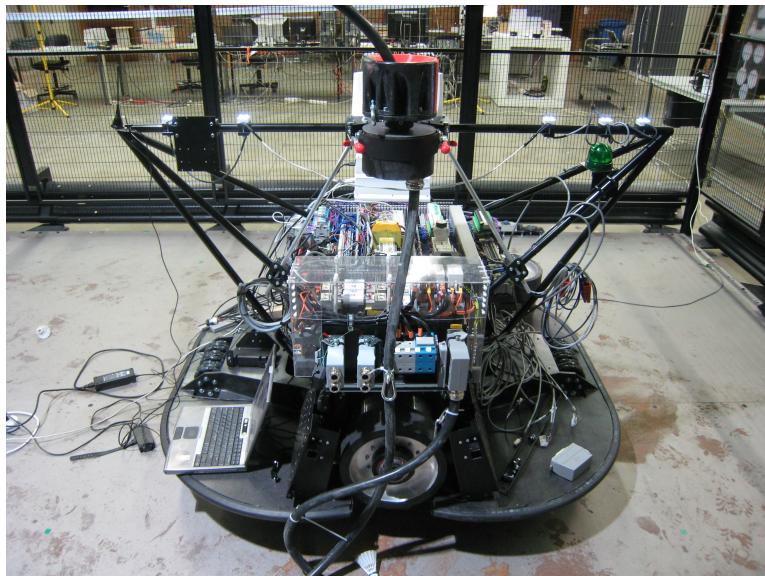


Figure 6.1: The automated guided vehicle of the case study.

The AGV is designed to transport payload on a specific trajectory between a set of locations. The drivetrain is fully electrical, using a battery for energy storage and two electric motors driving two wheels. Being a complex mechatronic system, the requirements of the AGV are specified by stakeholders of the different involved domains, such as

- mechanical requirements: sufficient room on the vehicle to place payload;
- control requirements: following the defined trajectory with a given maximal tracking error;
- electrical requirements: autonomous behavior, defined as the number of times that it needs to be able to perform the movement before needing to recharge;
- product quality requirements: the previous requirements should be achieved at a minimal cost.

Figure 6.2 shows the conceptual geometric design of the AGV. The design team chose a circular platform, with two omniwheels in addition to the two driven wheels.

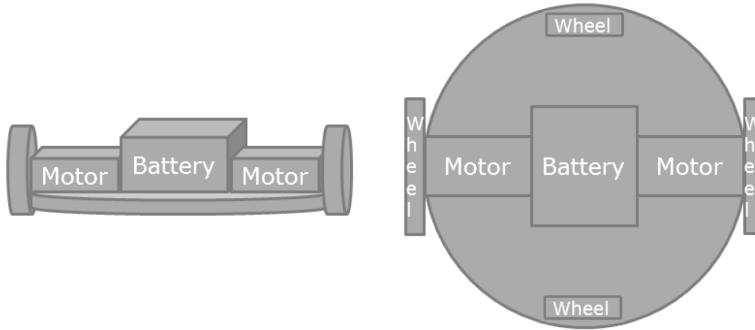


Figure 6.2: Front and top view of the conceptual design of the AGV.

The design process needs to determine the sizing of the different components (motors, battery, platform) and tune the controller. This process is decomposed into multiple dependent design steps, such as motor selection, battery selection, platform-, controller-, and drivetrain design. The process requires an interplay between different domain-specific engineering tools, such as CAD tools for platform design, Simulink and Virtual.Lab Motion for multi-body simulations employed during controller design, AMESim for multi-physical simulations during drivetrain design. Motor and battery selection activities use databases maintained in Excel files. Since these tools work with different modeling formalisms, reasoning over the consistency of the system as a whole properties poses a complex problem to overcome. By explicitly modeling linguistic and ontological properties and associating them with the engineering activities, patterns of inconsistencies can be identified and handled.

## Naïve solution using Matlab

A naïve solution has been given by our industrial partner to formalizing the process for automation purposes. To this end, a Matlab script has been developed to capture the

essential steps of the process. The outline of this script is shown in Listing 3. Listings 22-36 of Appendix D document this script in details.

The script first initializes the variables used throughout the process, such as initial conditions of simulation parameters, and contents of catalogues to choose components from. Subsequently, seven iterations of the engineering process are hard-coded.

---

### **Listing 3 Outline of the original design process encoded in Matlab.**

---

```

1 %%Initial conditions, variables, etc
2 designSteps = containers.Map();
3     % Motor selection design step
4 motorSelection.parameters = {'Pdes','Peff','m','r','l'};
5 motorSelection.inputs = {'Pdes'};
6 motorSelection.outputs = {'m','l','r','Peff'};
7 motorSelection.analytical = motorAnalytical;
8 motorSelection.functionCall = '[Peff,m,l,r] = MotorSelectionF(Pdes)';
9 motorSelection.data = containers.Map();
10 designSteps('motorSelection') = motorSelection;
11     % Battery selection design step
12 batterySelection.parameters = {'Cdes','Ceff','m','l','w','h'};
13 batterySelection.inputs = {'Cdes'};
14 batterySelection.outputs = {'m','l','w','h','Ceff'};
15 batterySelection.analytical = batteryAnalytical;
16 batterySelection.functionCall = '[Ceff,m,l,w,h] = BatterySelectionF(Cdes)';
17 batterySelection.data = containers.Map();
18 designSteps('batterySelection') = batterySelection;
19 %%(...)
20
21 %%Iterations of the engineering process
22
23     % ITERATION 1
24 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
25 motorSelectionPoint = [motorSelectionPoint;call(motorSelection,motorSelectionPoint)];
26 add(motorSelection.data,motorSelectionPoint);
27
28 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
29 batterySelectionPoint=[batterySelectionPoint;call(batterySelection,
    ↪ batterySelectionPoint)];
30 add(batterySelection.data,batterySelectionPoint);
31
32 assignInputs('mechDesign');
33 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
34 add(mechDesign.data,mechDesignPoint);
35 %%(...)
36
37     % ITERATION 2
38 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
39 motorSelectionPoint = [motorSelectionPoint;call(motorSelection,motorSelectionPoint)];
40 add(motorSelection.data,motorSelectionPoint);
41
42 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
43 batterySelectionPoint=[batterySelectionPoint;call(batterySelection,
    ↪ batterySelectionPoint)];
44 add(batterySelection.data,batterySelectionPoint);
45 %%(...)
46
47     % ITERATIONS 3-7
48 %%(...)
49
50 %%Function definitions
51 %%(...)

```

---

It is far from obvious from this description, but every iteration consists of the same activities. These are the following.

**Motor selection** Selects the first appropriate motor from a list (catalogue), based on the current precondition of the *desired minimal power*. (See Lines 2-4 in Listing 24.)

**Battery selection** Selects the first appropriate battery from a list (catalogue), based on the current precondition of the *desired minimal capacity*. (See Lines 6-8 in Listing 24.)

**Mechanical design** Applies the selected motor and battery to the abstracted mechanical design and simulates the resulting model. The inputs to this activity are the masses, geometry and positions of the newly selected components. The result of this activity is the inertia matrix of the resulting vehicle. The inputs and outputs of the activities are shown in Lines 27-70 of Listing 22 of Appendix D. Specifically, the input for the Mechanical design activity are shown in Lines 48-56.

**Control optimization** The control should be optimized in order to fulfill the AGV's mission criteria. The inertia matrix is taken as an input and as an output, the required power of the motor and the torque profiles of the wheels are produced.

**Electrical simulation** The electrical simulation takes the torques and simulates the electrical model to check if the battery can provide enough energy at every point of the simulated trajectory to the motor in order it to generate the appropriate torques. As a result, the feasibility of the configuration is obtained. If the electricity or the power are not sufficient to keep the vehicle on the desired trajectory, the process will start a new iteration.

**Select next point** To determine how far the current solution is from a feasible/optimal one, a response-surface methodology (RSM) is used. We abstract from this step, as it is not necessary for our purposes. It is, however, included in the case study to demonstrate full correspondence.

## Structure of this chapter

In Section 6.2, we demonstrate that our approach is capable to fully reproduce the engineering process previously captured as a sequential Matlab script by our industrial partner. Subsequently, in Section 6.3, we re-model the whole process leveraging the capabilities of our approach, optimize the process for consistency, enact it while ensuring service integration, and manage inconsistencies during the enactment.

## 6.2 Modeled reproduction of the hard-coded process

In our first case, we take the Matlab script discussed at the end of Section 6.1, and re-model it so that the semantics will still be mainly given by Matlab. The aim of this exercise is to demonstrate the ability to facilitate one-to-one correspondence between an FTG+PM process model and a hard-coded Matlab script.

Figure 6.3 shows the process model of the case. To reproduce the original Matlab-encoded workflow, only the PM and the typing FTG are needed.

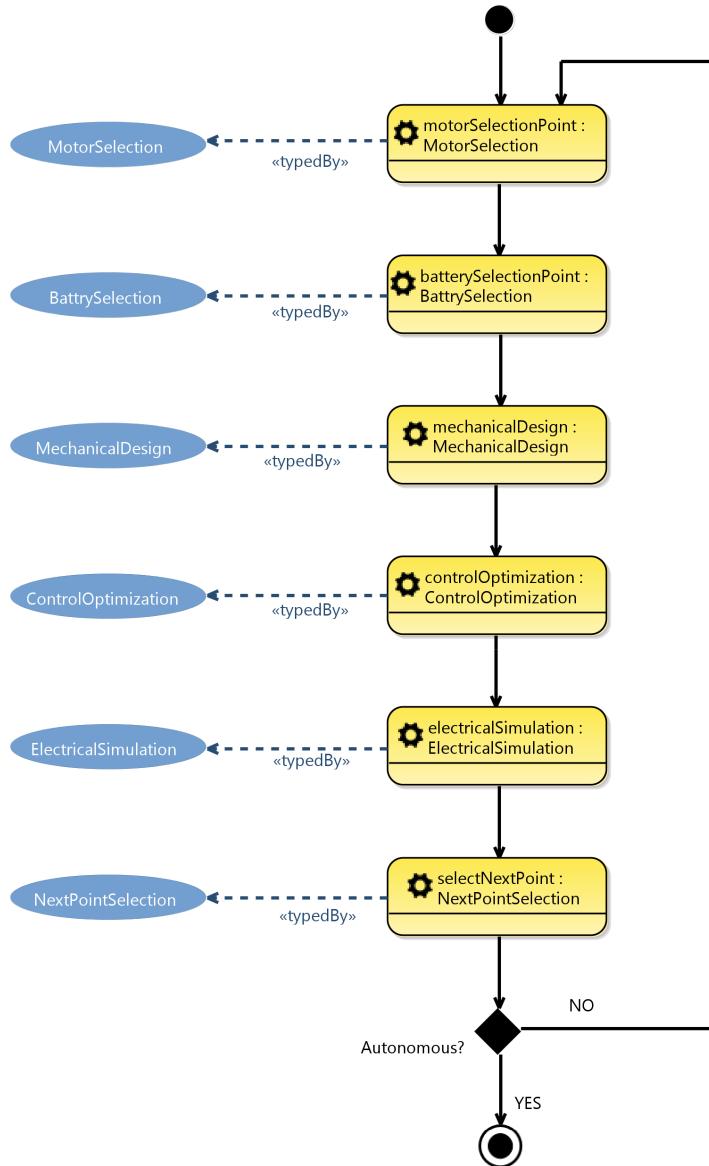


Figure 6.3: The FTG+PM of the case study.

The decision point after the elementary engineering operations is called *Autonomous?*, and it represents the fact that the vehicle is able to carry out its mission.

The artifacts flowing through the process would be attributes, properties and simulation traces, but in this case, it is Matlab that takes care of the actual execution semantics. After having modeled the structure of the process, the correspondence with the appropriate parts of the Matlab code has been facilitated and provided to the transformations of the model shown in Figure 6.3. Our tool allows associating Matlab, Java and Python scripts with

the transformation definitions in order to provide semantics. Figure 6.4 shows that the semantics of the *motorSelectionPoint* activity are given by the *motorSelection.m* script. The script is, in fact, associated with the *MotorSelection* transformation, so any activity typed by this transformation will be given semantics by the *motorSelection.m* script. To tailor the script to the specific activity, execution parameters can be specified for the activity, and used with a specific syntax in the scripts. (For Matlab, it is *args[i]*, where *i* represents the index of the parameter in the parameter list. The name and the value of the parameter can be obtained using the *args[i].name* and the *args[i].value* calls, respectively. The list of parameters is a comma-separated list of semicolon-separated key:value pairs.)

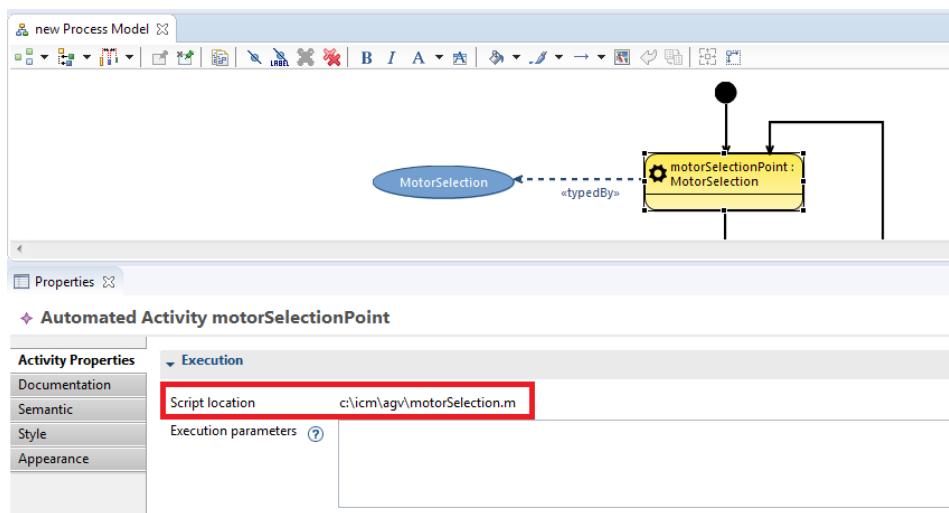


Figure 6.4: Typed activity with the Matlab script *motorSelection.m* providing the execution semantics.

The *motorSelection.m* script is fairly simple, as shown in Listing 4.

---

**Listing 4** Matlab script providing semantics for the *MotorSelection* transformation.

---

```
1 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
2 call(motorSelection,'motorSelectionPoint');
3 add(motorSelection.data, motorSelectionPoint);
```

---

After having lifted all the activity-related parts of the script, the rest of the script is treated as a library, which provides the functionality to the activities.

Finally, as Figure 6.5 shows, the built-in functionality of generating Matlab scripts, can be invoked via the main menu available from the process modeling canvas.

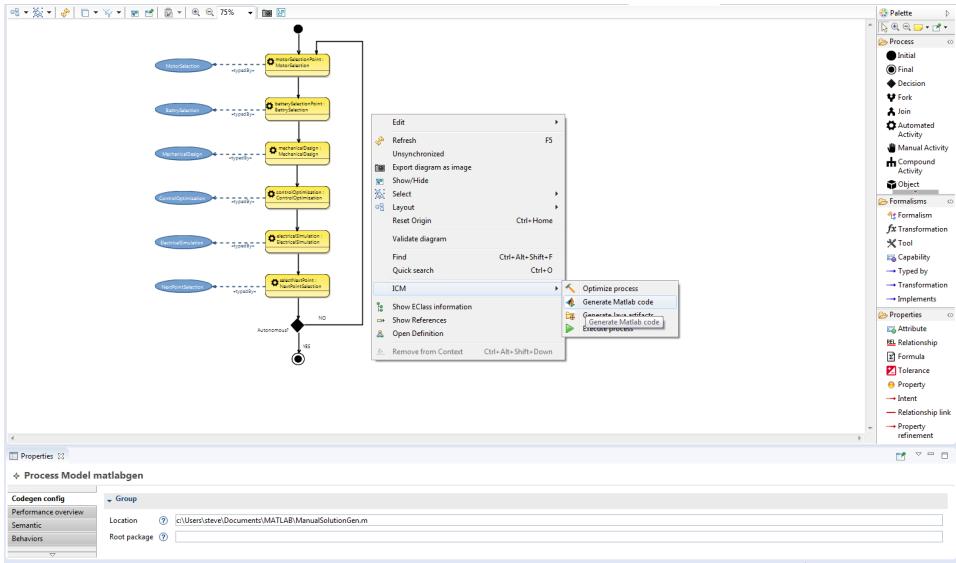


Figure 6.5: The built-in functionality for generating Matlab scripts shown in the menu.

## 6.3 Modeling the case using our approach

In our second case study, we take previously elaborated case and improve its process model in order to demonstrate the capabilities of our approach. The process model has been created using the domain-specific knowledge from our industrial partners. We aim at demonstrating all the four important stages of our approach: (i) process modeling; (ii) modeling of the attributes, properties and constraints; (iii) off-line inconsistency management; and (iv) enactment and on-line inconsistency management.

### 6.3.1 Process modeling

First off, we have to model the process. We improve the process so that the *battery selection* and the *motor selection* activities are carried out *in parallel*. This will lead to new types of inconsistencies which have to be managed.

Figure 6.6 shows the PM of the case with the activities and artifacts; and Figure 6.7 shows the FTG typing the process. In the following, we elaborate on the elements of the process model by walking through its activities.

#### ***setUpMotorDB* and *setUpBatteryDB***

Throughout the process, the previously established motor and battery databases are used to choose components from automatically, based on various conditions.

To this end, first the motor database is set up in the *setUpMotorDB* activity. The activity realizes the *MotorDBSetup* transformation, which initializes a *Matlab* structure called *motorDB* out of the manually defined list of motors.

The *setUpBatteryDB* activity serves a similar purpose, but for the battery components. Instead of taking a manually defined list, the activity takes a *csv* list of battery components, called *KokamLargeBatteries*, and creates a *Matlab* structure out of it, called *batteryDB*.

Both of the steps are given semantics by *Matlab*. The associated code snippets are listed in Appendix D. Listing 37 shows the Matlab code for the *setUpMotorDB* activity; Listing 38 shows the Matlab code for the *setUpBatteryDB* activity; and Listing 39 shows the original source of the battery DB: the *KokamLargeBatteries* collection.

### ***setInitialConditions***

The initial conditions of the process are set in the *setInitialConditions* activity. This means opening a *Matlab* workspace and defining and initializing the parameters (variables, in technical terms) used later.

The original process covers these parts in Lines 14-70 of Listing 22 and Lines 1-23 of Listing 23 of Appendix D.

### ***selectBattery* and *selectMotor***

Subsequently, the actual engineering commences, and that, in a parallel fashion. The battery and the motor are selected concurrently, in the respective *selectBattery* and *selectMotor* activities. Both of these activities are typed by the *ComponentSelection* transformation, which takes a *Matlab* struct and selects the first appropriate component out of it. In the case of the *selectBattery* activity, it is the *batteryDB* that is being used for the selection; in the case of the *selectMotor* activity, it is the *motorDB*. The selected components, both the battery and the motor are then stored in the *Matlab* workspace as variables, under the name *selectedBattery* and *selectedMotor*, respectively.

The Matlab scripts giving semantics to the *selectBattery* and *selectMotor* activities are listed in Listing 40 and Listing 41 of Appendix D, respectively.

The process joins at this point, meaning, both the previous activities have to be concluded, before proceeding.

### ***simulateMechanicalModel***

The first step after selecting the two components, is the mechanical simulation, carried out by the *simulateMechanicalModel* activity. The *mechanicalModel*, again, is a *Matlab* structure in this case. The model is a lumped parameter model of the actual mechanical design, and can in more detailed scenarios, it is usually captured in formalisms like *CAD*. In this case, the mechanical model serves the purpose of calculating the inertia of the newly configured vehicle, hence the simplified model.

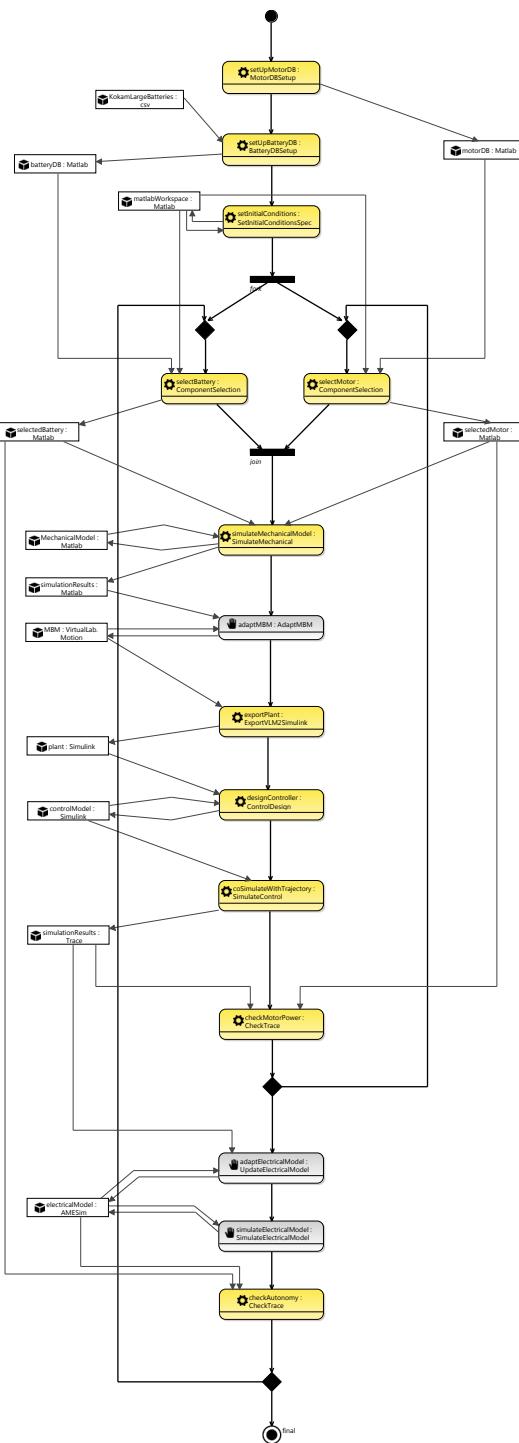


Figure 6.6: The PM of the case.

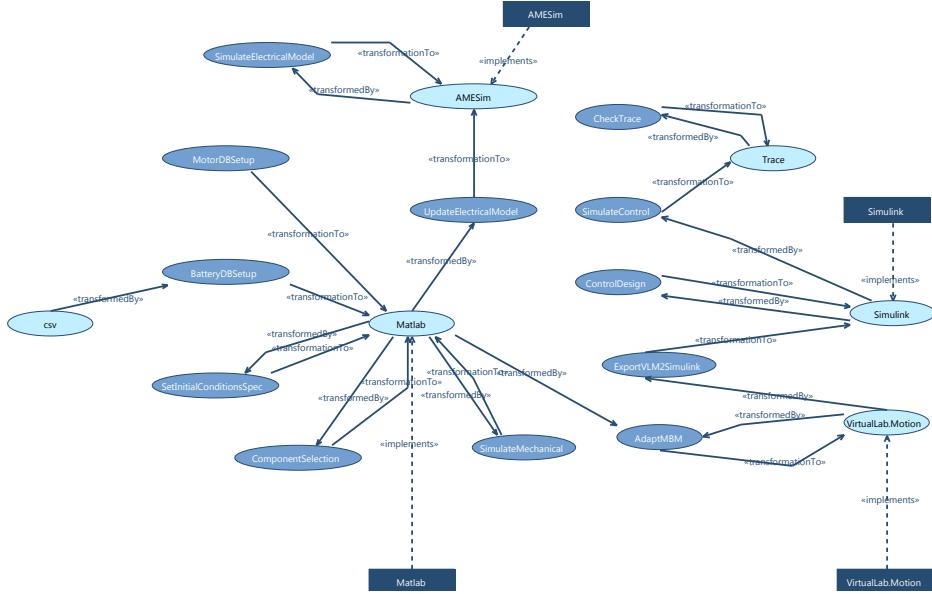


Figure 6.7: The FTG of the case.

The model is a collection of seven variables: the masses of the battery and the motor; the length-width-depth dimensions of the motor approximated by a cuboid shape; and the radius and height dimensions of the battery approximated by a cylinder shape.

The result of the simulation is the total mass and the three components of the 3D inertia matrix. The results are captured in the *simulationResults* artifact, and are persisted in *Matlab* variables.

The Matlab script giving semantics to the activity is listed in Listing 42 of Appendix D.

### **adaptMBM**

In the next step, the first *manual* activity of the process is being carried out. The multi-body model consists of a multitude of bodies, joints, and constraints, forces and torques, and sensors. [125]

The activity takes the previously obtained simulation results, and updates the multi-body model in the VirtualLab.Motion tool. [179] This multi-body modeling tool allows exporting a plant model compatible with Simulink for control design purposes. This is what is happening in the subsequent steps.

***exportPlant***

The *exportPlant* activity exports a plant model from the updated multi-body model in order to carry out the control design.

The engineering of the control logic is carried out in Mathwork's Simulink [?], and therefore, the *plant* model is typed by the *Simulink* formalism.

The result of this activity is shown in Listing 43 of Appendix D.

***designController***

The *designController* activity is an automated one. The previously design control model is updated with the new plant model and the simulation is carried out. As a result, the *controlModel* is produced in *Simulink*.

***coSimulateWithTrajectory***

The expected trajectory of the vehicle is then taken into account and a co-simulation of the *controlModel* and the *multi-body model* (referred as: the controlled multi-body model) is carried out.

The results of this activity are (i) the maximal power of the motor required for the control logic; and (ii) the average torques on the two wheels. These parameters are used later to decide if the motor and the battery were sufficient (in terms of power and capacity, respectively) for the eventually correct configuration.

***checkMotorPower***

The required power of the motor is checked against the maximal power of the motor selected in the *selectMotor* step. If the required power can be provided by the motor, the process continues, otherwise it falls back to the *selectMotor* activity.

***adaptElectricalModel***

Should the selected motor be able to provide the required power, the electrical capabilities must be checked as well. This means, it should be investigated if the selected batteries can provide the sufficient capacity to power the motors and generate the sufficient torques prescribed by the control logic.

First, the electrical model is updated. The electrical model is stored in the AMESim tool [168].

### ***simulateElectricalModel***

The simulation of the electrical model takes the torques obtained in the *coSimulateWithTrajectory* activity. As a result, the desired capacity is obtained.

### ***checkAutonomy***

Finally, the desired capacity is compared to the actual capacity of the battery. If the battery can provide the desired capacity, the vehicle is deemed autonomous and the process concludes. Otherwise the process falls back to the *selectBattery* activity.

## 6.3.2 Attributes, properties, constraints

Next up, the attributes, properties and constraints have to be modeled in order to later carry out the inconsistency management. Figure 6.8 shows the PM augmented with the attributes, properties and constraints. The figure is extremely cluttered due to the heavily interconnected model. The provided tool helps organizing the model into various views and within the views, into various layers. This helps with visualizing the parts of the model which are relevant to specific stakeholders.

Here, we outline the main points of the model.

### 6.3.2.1 Attributes

The attributes are denoted by the red rectangles on the right side of the process. The dashed lines represent intents of an activity to an attribute.

***desiredPower, desiredCapacity – actualPower, actualCapacity*** The *desiredPower* and *desiredCapacity* attributes are used to decide whether the power of the chosen motor, and the capacity of the chosen battery is sufficient or not, respectively.

The *desired-\** attributes set in the *setInitialConditions* activity (hence the *modify* intent), and later modified by the simulation activities: *coSimulateWithTrajectory* and *simulateElectricalModel*, respectively.

The counterparts of these attributes are the *actualPower* and *actualCapacity* attributes, representing the actual characteristics of the chosen components.

***supportTime*** The *supportTime* attribute is used to derive the capacity from the current. Usually, the following equation is used for this purpose:

$$\text{capacity} \geq \int \text{currentDrawn}(t) dt.$$

But we simplify this calculation by abstracting to a steady current.

**Masses: platform, motor, battery, magnet, total** Masses pertaining to the various components are captured in the *\*-mass* attributes. When calculating the total mass, the sum of the platform, the motors (2 of them), the battery and the magnets (2 of them) are take into consideration.

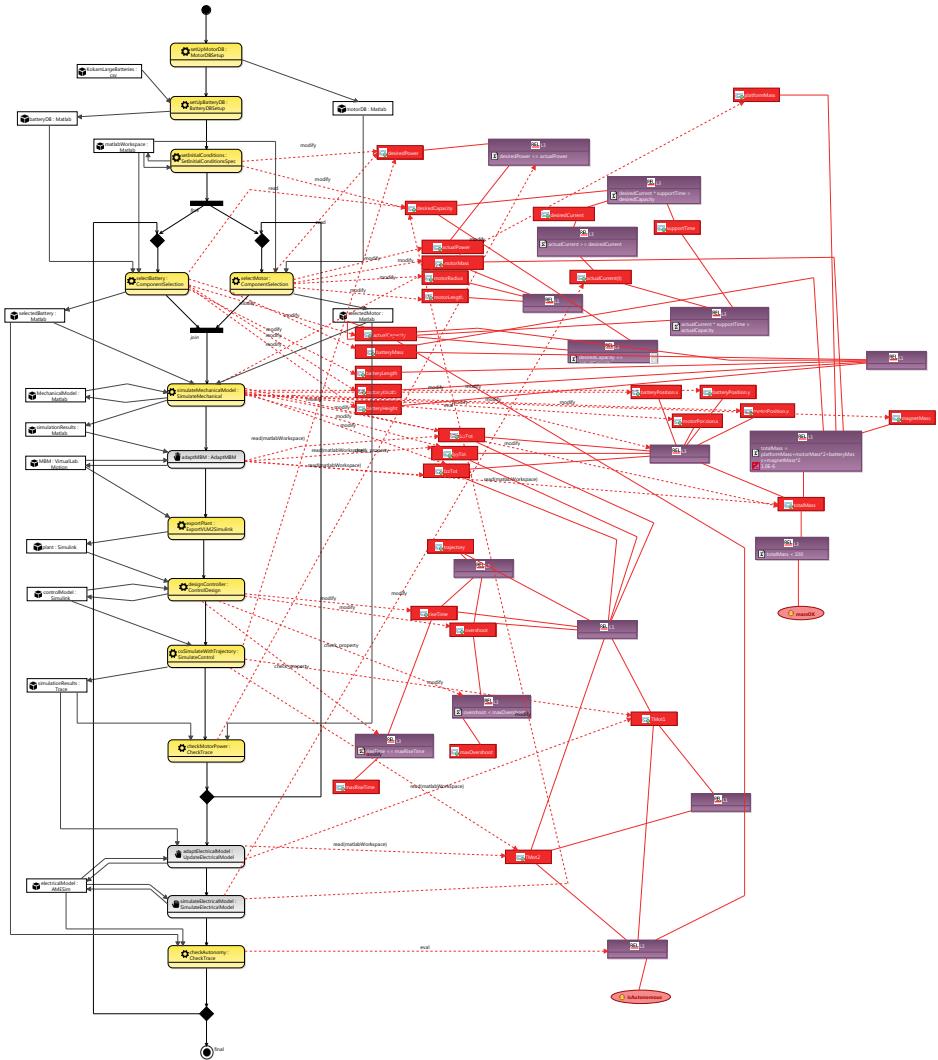


Figure 6.8: The augmented PM of the case.

### 6.3.2.2 Constraints

Constraints are defined using the previously modeled attributes.  $\mathbb{L}3$  relationships are paramount to the efficient change propagation and early inconsistency detection in the on-line phase. It is, however, the  $\mathbb{L}1$  and  $\mathbb{L}2$  relationships that prevail in our case.

Here, we look at some of the demonstrative examples.

**$\mathbb{L}3$ : *actual-\**  $\geq$  *desired-\**** An important  $\mathbb{L}3$  constraint type is the one that defines that the *actualPower* should be at least as high as the *desiredPower*; and that the *actualCapacity* should be at least as high as the *desiredCapacity*. These are the relationships that govern the two loops of iteration in the process, when the *checkMotorPower* and the *checkAutonomy* activities are executed, respectively.

The rules are as follows:

- $\text{desiredPower} \leq \text{actualPower}$
- $\text{desiredCapacity} \leq \text{actualCapacity}$

**$\mathbb{L}3$ : *capacity = current \* supportTime*** As elaborated previously, we assume a constant current being drawn from the battery and hence, we approximate the capacity required to meet the support time with the above formula.

There are two variants of this expression: one for the desired and one for the actual capacity.

**$\mathbb{L}3$ : *totalMass – with tolerance*** The total mass is the sum of the various components which are characterized by a mass. This entails the platform, two motors, the battery and two magnets. The model suggests that  $\text{totalMass} = \text{platformMass} + \text{motorMass}^*2 + \text{batteryMass} + \text{magnetMass}^*2$ .

In this specific case, the evaluation of the equation is also supported with a tolerance type of a parameter. We are willing to tolerate an additive error in the 1.0E-6 order of magnitude. The machinery for the evaluation of the constraint will consider this error margin when warning for potential inconsistencies, hence implementing the tolerance principles discussed in Section 5.7.5.1.

### 6.3.3 Properties

For demonstrative reasons, we also provide two properties. Both the *massOK* and *isAutonomous* properties are derived from a performance value [203] by mapping to the *B* domain. Should the associated equation not hold, the property is considered unsatisfied; should the equation hold, the property is considered satisfied; otherwise the satisfaction of the property is deemed inconclusive.

### 6.3.4 Resources

At this point, we assume that the pool of resources always satisfies the resource demands, so we intentionally model the resources with this in mind. Figure 6.9 shows our resource model with the typing and availability information. The top three elements represent the human resources: the mechanical, electrical and control engineers, one available from each. The bottom five elements represent the tools and licences required to carry out the activities of the process. It is only the *selectBattery* and *selectMotor* activities being executed in parallel, and these only overlap in using one *Microsoft Excel licence* each. Thus, two licences are required to keep the process parallel. Should there be only one licence, the parallelized process would be transformed into a sequential one, in which the *selectBattery* and *selectMotor* activities are situated in a sequential order.

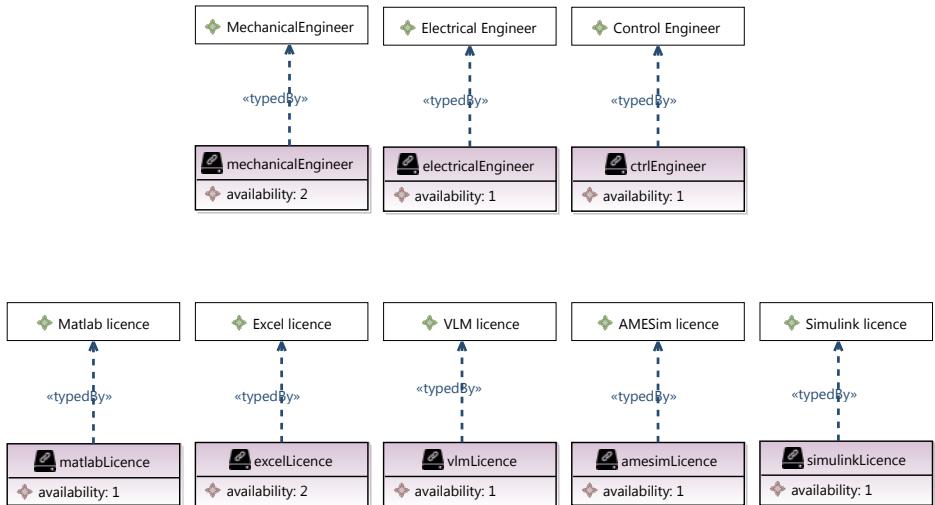


Figure 6.9: The resource model of the case.

### 6.3.5 Costs

Finally, the costs are modeled. Our approach supports a multi-objective approach to the automated optimization of processes. In this case, this would mean providing multiple cost metrics for the process and arbitrarily selecting one process candidate from the Pareto-front [48].

In order to keep the case study fairly simple and demonstrative, we operate with one type of cost: the execution time of the various activities. Table 6.1 lists the chosen execution times.

(Due to the inconvenient layout of the cost matrix provided by the tool, we only show a table at this point. The tool, however, allows modeling the costs efficiently, but the visualization is not suitable for static documents.)

Activity	Execution time [hours]
setUpMotorDB	4
setUpBatteryDB	4
setInitialConditions	0.1
selectBattery	0
selectMotor	0
simulateMechanicalModel	1
adaptMBM	4
exportPlant	0.1
designController	8
coSimulateWithTrajectory	2
checkMotorPower	0.1
adaptElectricalModel	8
simulateElectricalModel	4
checkAutonomy	0.1

Table 6.1: Execution times of the activities.

### 6.3.6 Off-line inconsistency management

We use the default library of inconsistency patterns and management patterns for the off-line phase. We do so, because the defaults have been developed while investigating this very case study. Therefore, the default libraries are the most appropriate ones for this particular problem. Additional patterns can be formulated for different problems. (Caveat: there is no user-friendly interface on the tool for carrying out such an activity. Such a step should be accomplished by modifying the graph queries and transformations in the source of the tooling, and recompiling the tool.)

The functionality for the off-line inconsistency management can be invoked via the main menu available from the process modeling canvas, as shown in Figure 6.10.

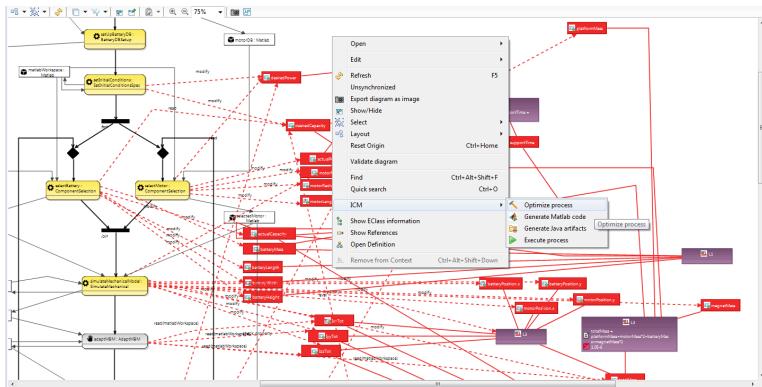


Figure 6.10: Invoking the optimization mechanism on the PM.

### The optimized process

The optimization results in a slightly transformed process shown in Figure 6.11. To avoid a cluttered figure, we show only the essential attributes of the model. The rest of the attributes is left intact, as shown in Figure 6.8.

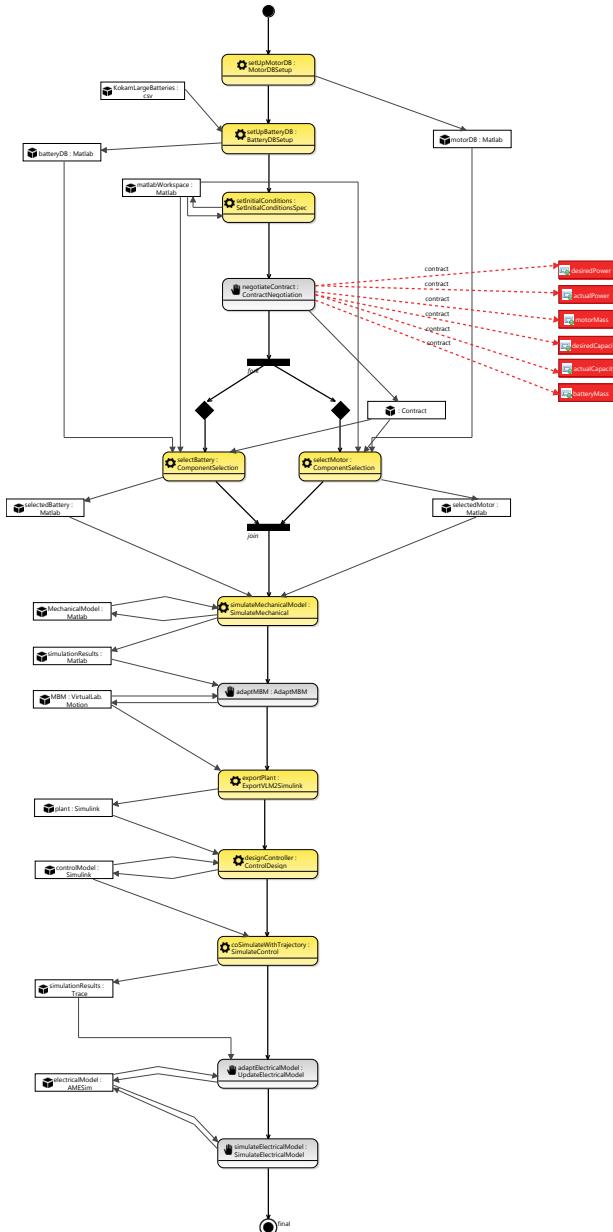


Figure 6.11: The optimized PM of the case.

The inconsistency manager identified the potential deviation between the pairs of *desiredCapacity-actualCapacity*, and *desiredPower-actualPower* as the source of the potential inconsistency in this model. The model has a built-in mechanism for identifying such inconsistencies, namely: the two feedback loops. This structure has been identified as an inconsistency management pattern and evaluated against other management patterns. As witnessed by Figure 6.11, the cost simulations deemed the feedback loops potentially very costly, as opposed to the pattern of using design contracts. Indeed, the *contractNegotiation* activity is required only once for the particular inconsistency [204]. The linchpin of the decision, of course, is the estimation as of how much time the *contractNegotiation* activity requires. In our experiments, we chose the realistic 8 hours for such an activity.

As a consequence, the proposed managed process features a contract negotiation phase, which results in a design *Contract*. The contract imposes *contract* intents on the following attributes: *desiredPower*, *actualPower*, *desiredCapacity*, *actualCapacity*, *motorMass*, *batteryMass*. By doing so, the *contractNegotiation* activity conceptually guarantees no inconsistency will be introduced through these attributes.

As for the performance: the new process shows an improvement of 80%. (A decrease from about 220 hours to about 43.) This figure, however, is heavily influenced by two facts. First, the cost calculation is sensitive to slight differences in the time estimations. Second, we implicitly assumed that negotiating a design contract is an efficient and feasible activity, which is far from granted in nowadays engineering state-of-the-practice.

The caveat of this step is obvious: we do not model the internal of the design contract. But this has not been the goal anyways. Modeling design contracts is a research field of its own, and its foundations have been just recently published by Vanherpen [204].

### 6.3.7 On-line inconsistency management

Finally, the process is enacted and the on-line inconsistency management mechanism kicks in. To ensure tool interoperability, two steps have to be taken before enacting the process.

First, the formalisms used throughout the FTG+PM have to be associated with *tools*. We did this previously, as shown in Figure 6.7. The dark blue rectangles represent the tools implementing the various formalisms. Currently, there is a one-one relation between formalisms and tools, meaning: tools supporting multiple formalisms have to be added in multiple instances. (This design decision has been made to short-circuit the non-trivial problem of multi-formalism tools and their relations to each other.)

Second, the attributes may exist in external tools under different aliases. Sometimes, attributes are not even persisted in a serialized model or workspace, but are derived and calculated on-the-fly [153]. To overcome this problem, our tool offers a table-based alias-editor for the attributes. Figure 6.12 shows four attribute-alias pairs. For example, the *actualPower* attribute in the *motorDB* artifact is referred as *Peff*. Multiple aliases can be listed for a single attribute, and the machinery will attempt to find one of these names in the artifact at hand to execute the activity. Specifically, derived features captured as EMF-IncQuery queries [153] can be referred with the following syntax: @query : [name].

actualPower	motorDB:Peff
actualCapacity	batteryDB:Ceff
batteryMass	
desiredPower	matlabWorkspace: PdesV
desiredCapacity	matlabWorkspace: CdesV

Figure 6.12: Alias table for some of the attributes.

Now the process is ready to be enacted. The functionality for enacting the process can be invoked via the main menu available from the process modeling canvas, as shown in Figure 6.13.

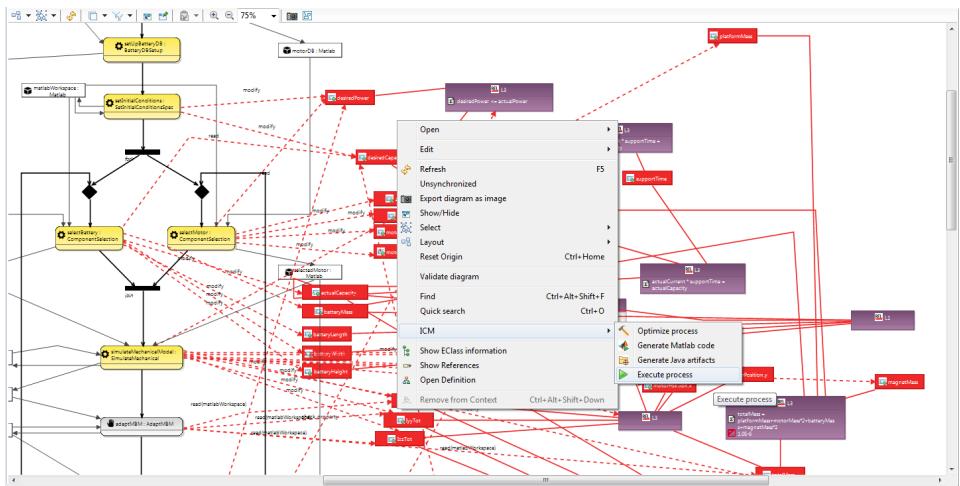


Figure 6.13: Invoking the process enactment mechanism on the PM.

The enactment of the process is managed through a process management console. The commands are documented in Section 7.5. The automated activities are attempted to be executed in a headless fashion, i.e. without actually showing the UI of the engineering tool. This is especially convenient when the tool take lots of resources to boot up in a UI mode, which is the case, e.g. for AMESim.

AMESim provides a reasonable scripting interface through C and Python. We opted for the latter one. In case of Matlab, we opted for the managed Java API.

In terms of processing overhead, there is a strong distinction between *querying* a tool for information and *modifying* an artifact in a tool. To this end, we distinguish between query type of requests, and execution type of requests. The modules covering these functionality are listed in Listing 44, Listing 45, Listing 46, Listing 47 and Listing 48 of Appendix D.

Figure 6.14 shows the process model that has been enacted, along with the attributes and capabilities relevant for the on-line inconsistency manager.

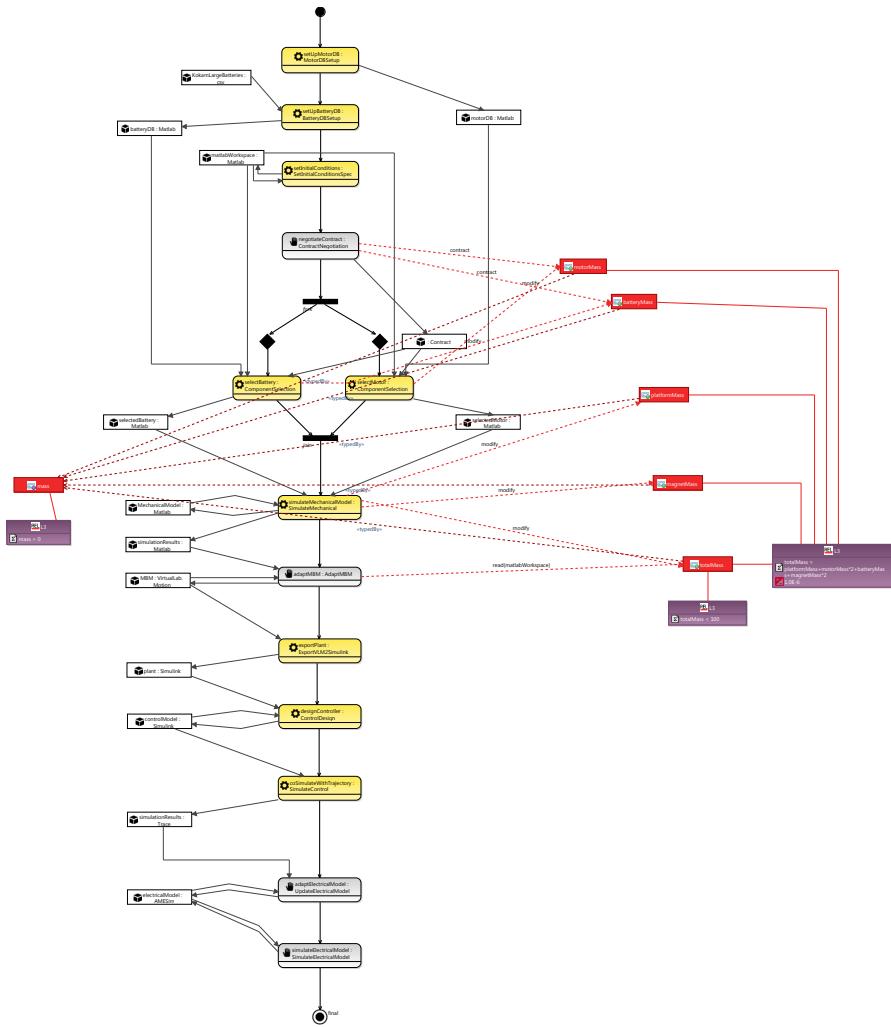


Figure 6.14: The enacted PM with the relevant attributes and capabilities of the on-line inconsistency management phase.

The inconsistency management example shown in Section 5.4 is actually taken from this case study, thus the reader is referred to that part of this thesis. Briefly: the early detection of inconsistencies is achieved by factoring in the ontological knowledge of the *mass* capability always being a positive number. This, along with the interconnections of the attributes and constraints brings valuable information for detecting inconsistencies even before they surface on the level of engineering models. The mechanism is described in Section 5.4.

# Chapter 7

## Prototype tooling

We support the methodology presented in this thesis with a prototype modeling and execution tool. The tool is available from <https://github.com/david-istvan/icm> under the Eclipse Public License 1.0.

### 7.1 Features

The main features of the tool are in accordance with the contributions of this research. That is, the tool supports

- modeling of engineering processes (Section 5.1);
- optimizing the modeled processes (Section 5.2);
- enacting the optimized process (Section 5.4);
- integrating external tools as services in the enactment process (Section 5.6).

### 7.2 Architecture

The tool has been mainly built on top of Eclipse technologies employing model-based techniques; with some additional features implemented in Python. The functionality supports the inconsistency management process shown in Figure 1.2. The general architectural overview of the tool is shown in Figure 7.1, with the main components distinguished by color coding. (Red: metamodels; green: off-line components; blue: on-line components; yellow: Eclipse-related productivity functionality.)

#### Metamodels

As we have employed a general model-based approach to develop the tooling, the functionality depends on the *Metamodels* providing formalisms for both the off-line and the

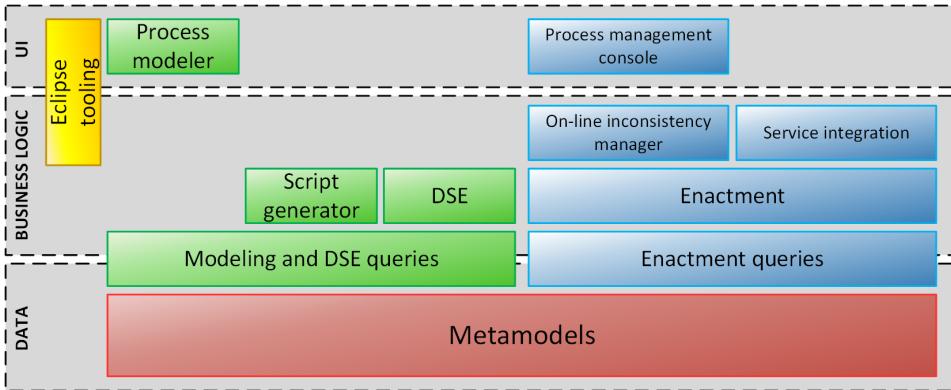


Figure 7.1: General architecture of the tool.

on-line components of the tool. The metamodels used all over the tooling are created using the Eclipse Modeling Framework (EMF) [56]. This includes metamodels for the pre-enactment functionality (process modeling and off-line inconsistency management), and for enactment-related features (enactment and on-line inconsistency management).

## Queries

Instances of the metamodels are manipulated through model transformations and queried via explicitly modeled queries. The off-line components use queries for *process modeling and DSE*, while the on-line components use queries for *enactment* purposes. For both of these, the VIATRA3 stack is used, which provides support for highly efficient, incremental model querying and executing transformations based on the queries.

## Off-line functionality

The off-line functionality consists of three components: the *Process modeler*, the *Script generator* for producing Matlab [124] scripts out of process definitions, and the *DSE* component implementing the off-line inconsistency management functionality. The *Process modeler* contributes solely to the UI layer of the tool, providing a rich, graphical user interface for modeling engineering processes.

## On-line functionality

The on-line functionality consists of four components. Out of these, the *Enactment* runtime, the *On-line inconsistency manager* and the *Service integration* components contribute to the business logic layer of the tool; while the *Process management console* is a UI element for controlling the enacted process.

### Eclipse-related productivity functionality

The Eclipse-related functionality consists of platform-specific features, such as

- project nature and project builder contribution,
- a project creation wizard,
- a model creation wizard.

These features improve the usability and the productivity of the tool.

## 7.3 Modeling of engineering processes

The support for modeling engineering processes has been implemented using Eclipse's Sirius framework [59]. The Sirius framework follows a model-based approach for developing various graphical user interfaces.

The tool we provide comes with two types of graphical interfaces: canvas-based modeling interfaces; and table-based modeling interfaces. Figure 7.2 shows a handful of these while modeling the FTG+PM of the running example, shown in Figure 5.7.

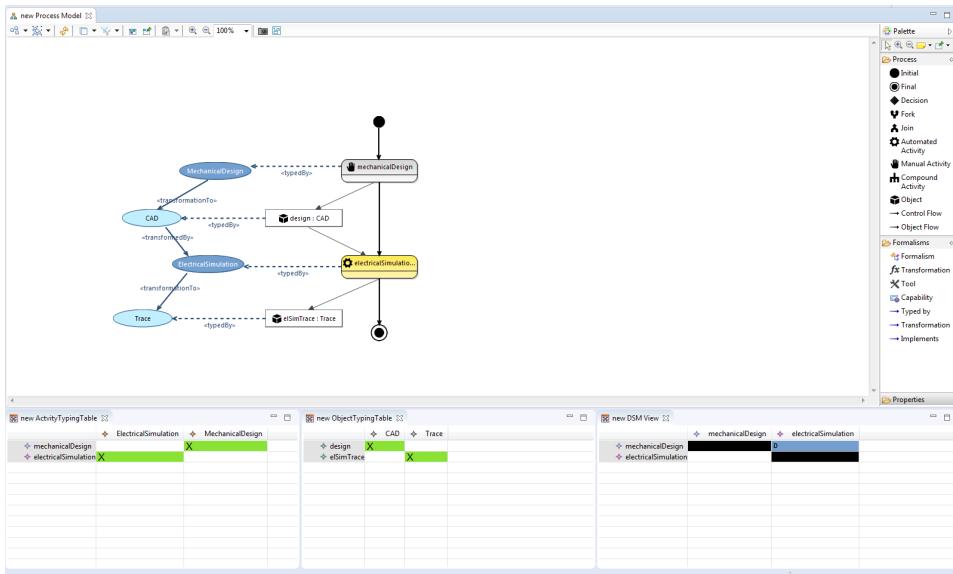


Figure 7.2: Various graphical components of the modeling UI.

The canvas-based central component allows the modeling the engineering process from various aspects. Dedicated interfaces allow modeling

- the FTG,
- the PM, and

- the resources

of the process model.

Apart from these, table-based components allow the modeling/viewing of other aspects of the engineering process, including

- artifact and activity typing,
- cost factors,
- attributes/properties and relationships,
- design-structure matrix derived from the process.

Design structure matrices (DSM) [64] are widely used in industrial settings to model dependencies of activities in complex engineering processes. While DSMs are specified manually and are not explicit about the reason of the dependency, our prototype tool provides a DSM-like view on processes that is able to distinguish between various types of dependencies, such as property, control and artifact dependencies.

### 7.3.1 Language specification

In this subsection, we disclose the elements of the graphical modeling formalism.

These language elements are available for the *Process Model* type, which is not to be mistaken with the bare PM. As shown in Figure 7.3, there are plenty other, mainly table-based representations for the Process model. As these table-based formalisms are straightforward to use, we omit them at this place.

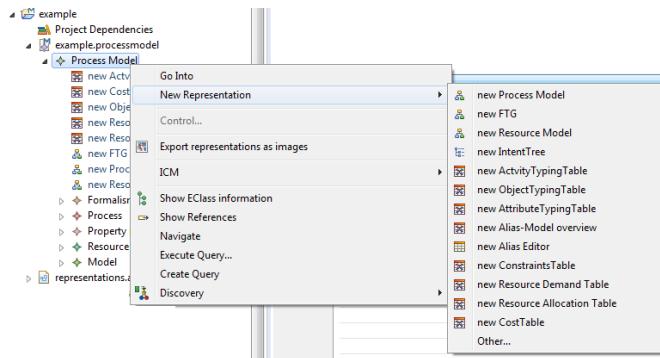


Figure 7.3: Representation types for the *Process Model*.

For the sake of completeness: as Figure 7.4 shows, there are two additional representations, specific to the PM: a table-based editor for the intents, and a DSM viewer [64].

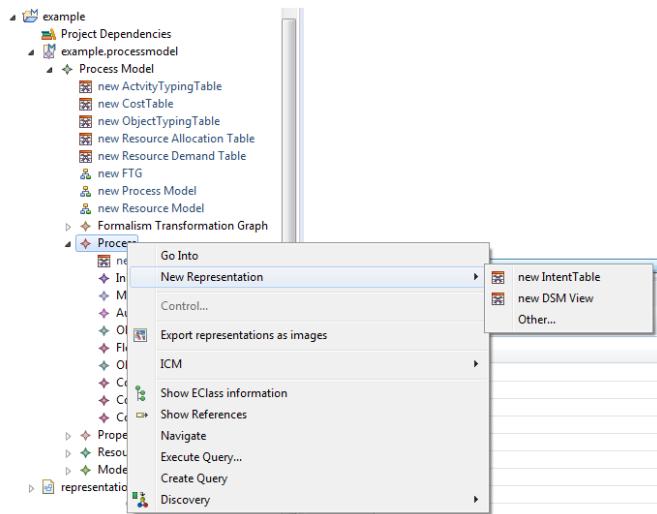


Figure 7.4: Representation types for the PM.

## Language elements

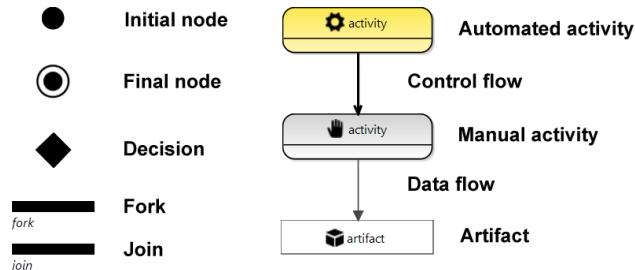


Figure 7.5: Elements for modeling the PM.

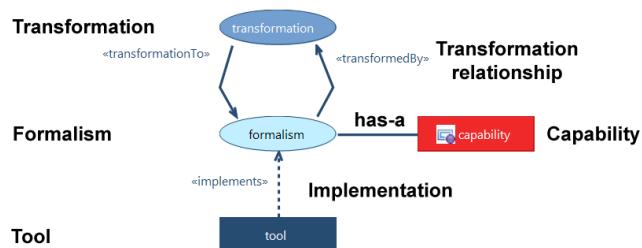


Figure 7.6: Elements for modeling the FTG.

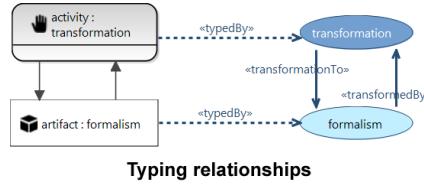


Figure 7.7: Elements for modeling the typing in the FTG+PM.

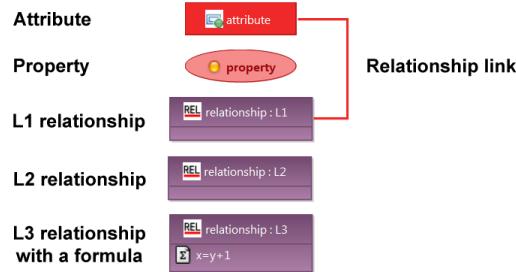


Figure 7.8: Elements for modeling the properties.

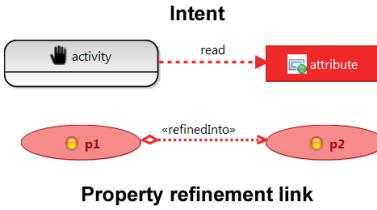


Figure 7.9: Additional elements for modeling the properties.

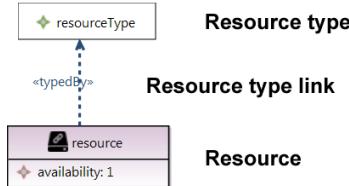


Figure 7.10: Additional elements for modeling the resources.

### 7.3.2 Eight important modeling patterns

Apart from the process-related patterns [187], there are patterns specific to our modeling formalism. We show eight frequently encountered ones out of these.

### Intent patterns

*Motivation:* Expressing the intent of an engineering operation w.r.t. the system.

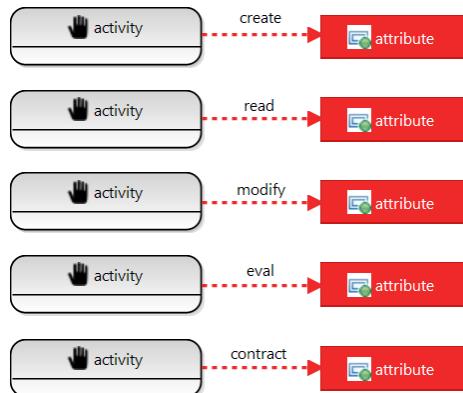


Figure 7.11: Intents.

### Model transformation patterns

Here, we follow the taxonomy by Mens and Gorp [129].

*Motivation:* Expressing the various techniques to transform a model.

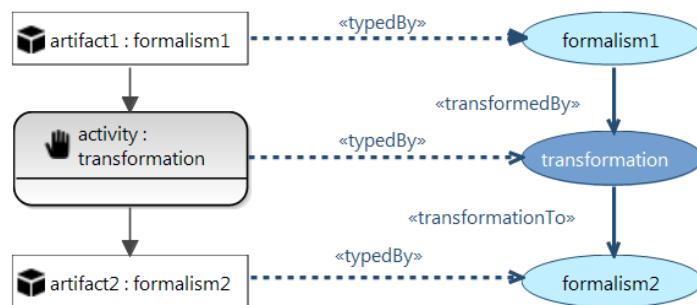


Figure 7.12: Exogenous model transformation.

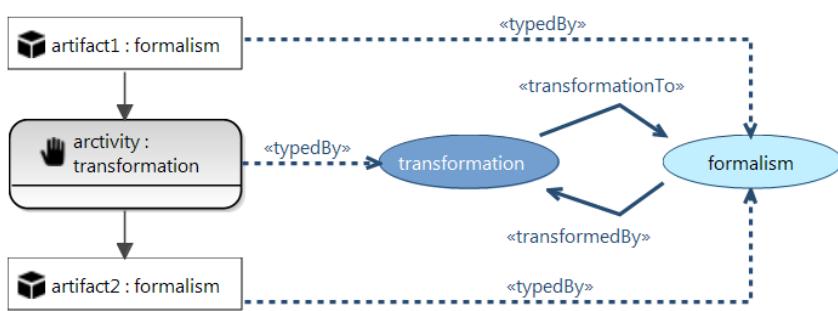


Figure 7.13: Endogenous model transformation.

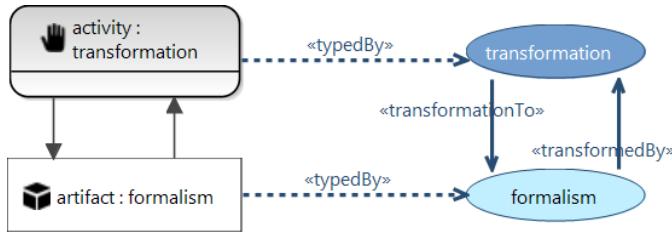


Figure 7.14: Endogenous in-place model transformation.

## Property patterns

Here, we follow the taxonomy by Vanherpen et al. [203].

*Motivation:* Expressing property information, e.g. refinement relationships, or obtaining the performance value of a property.

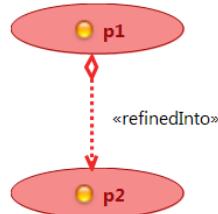


Figure 7.15: Property refinement.

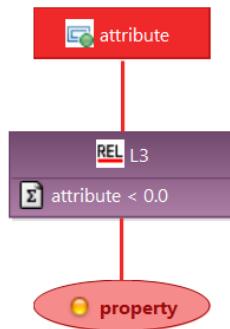


Figure 7.16: Performance value derived through an L3 relationship.

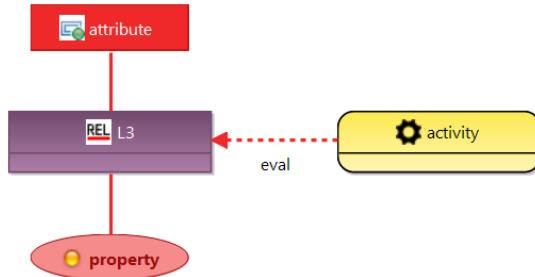


Figure 7.17: Explicit simulation of a performance value.

## Capability pattern

*Motivation:* Expressing the typing relationship between a property and a capability in order to use the latter information in inconsistency detection.

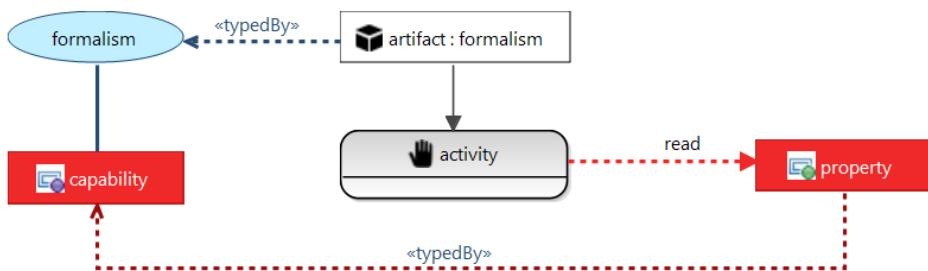


Figure 7.18: Property-capability typing.

### 7.3.3 Specifying executable scripts for automated activities

The tool allows specifying the automation logic for the automated activities of the process. This is achieved by associating a Matlab or Python script, or a Java code snippet with a transformation of the FTG. Every time an automated activity is being executed, the typing transformation's script or code is executed. Execution parameters can be specified for the specific activity, using a comma-separated list of semicolon-separated key-value pairs.

### 7.3.4 Code generation

Should the process model depict a process situated in a Matlab environment, and should all the automated activities be associated with Matlab scripts, the tool allows generating fully fledged, runnable Matlab source code. This feature has been developed to demonstrate one-to-one correspondence between the process models of our approach and the original engineering process of the AGV case, which has been developed in Matlab.

In general cases, Java skeletons are generated for automated activities without explicitly assigned execution semantics (Java code, Matlab scripts or Python scripts). Java was chosen for technological homogeneity w.r.t. the rest of the tooling that runs on the JVM as well.

### 7.3.5 Validation

In order to ensure the validity of process models, an on-the-fly validation support is added to the tool. The functionality has been implemented using the VIATRA3 incremental query framework that puts zero noticeable overhead on the modeling process as the validation is nearly instantaneous.

## 7.4 Off-line inconsistency management

The process optimization component has been built using the DSE engine of the VIATRA3 framework. The DSE engine takes an EMF instance model as an input and executes the exploration by applying VIATRA3 model transformations on it. This is guided by smart search algorithms so that the computation is feasible.

We opted for this specific framework because it enforces technological homogeneity and enables massive reuse of the models and model transformations already developed for the tool.

## 7.5 Enactment of engineering processes

The enactment engine is built by following models-at-runtime (M@RT) principles [26]. Such approaches aim at representing the prevailing state of the underlying system by explicit

models, and capturing execution semantics by model transformations.

Again, the model-based approach here is obvious, as are its benefits. Facilitating an enactment model (Figure A.7) is as easy as importing the previously defined FTG+PM related EMF models. The model transformations for the enactment semantics become very straightforward once a well thought out metamodel for the enactment is provided. The runtime transformations are shown in Listing 13.

In a similar spirit, incremental model transformations are used by Vogel et al [205] for efficient runtime monitoring. Song et al [170] introduce incremental QVT transformations for runtime models. Such approaches further underline the validity of our approach.

## The process enactment console

The enacted process is managed through a console interface. Its functionality is fairly complex, but with a usable textual interface. Below we list the available commands.

- atonce** The process executes as many automated activities as possible, at once.
- prepare [name]** Prepares the activity for enactment. This entails opening the engineering tool and loading the required artifacts.
- run [name]** Runs the activity, i.e. executes its semantics.
- finish [name]** Finishes the activity. This entails saving workspaces and artifacts and closing the tool.
- step [name]** Shorthand for *prepare+execute*.
- final** Executing the final step.
- exit** Exits the runtime.
- vardump** Prints the variables (attributes) of the process with their values.
- freevars** Prints the unbounded variables (attributes).
- freevarconstraints** Prints the constraints pertaining to the free variables along with their validity domains.

## 7.6 On-line inconsistency manager

During the enactment, the on-line inconsistency manager provides the necessary functionality to cope with the emerging inconsistencies. The centerpiece of the functionality is the *variable manager* that keeps track of the system variables, i.e. attributes, properties and relationships, and that, in a live and reactive fashion. As the process advances, the variables are continuously monitored upon changes of their own or their influencing attributes or properties.

The set of influencers is identified using the transitive closure through the relationship entities. This step is formalized and explained in Section 5.1.2.2.

The functionality is implemented using Python and EMF. The variable manager's data model is captured in EMF and maintained using incremental model transformations. The reasoning over the variables of inconsistencies is implemented in the Python-based framework for symbolic mathematics SymPy [176]. The detailed approach is discussed in Section 5.4.3.

## 7.7 Service integration

An MPM approach for explicit modeling of external service integration has been discussed earlier as a contribution of this research, but that approach is not integrated into the tool due to the technological challenges. Instead, direct support for Matlab and AMESim [168] is provided. The tool makes use of the managed APIs whenever possible. For Matlab, its Java API is used [123]. AMESim, however, does not provide an API for the managed JVM source code, only Python and C is supported. Efforts have been made to use the JVM-managed Jython [180] to facilitate the interaction with AMESim, but due to version incompatibilities, this proved to be a dead-end. The interfacing with AMESim is now solved by generating and executing Python scripts on-the-fly during the enactment.

## 7.8 Reflection and future directions

To reflect on the tooling itself, it can be stated that it served its purpose. It was *not* in the scope of this project to deliver a fully-functioning tool, yet, we have dedicated special efforts to it. We did so, because we believed a suitable prototype tooling makes it easier to demonstrate our scientific results to those outside the academia.

We have been heavily relying on model-based principles while developing the tool. Big parts of the functionality are implemented by the means of explicitly modeled internals: structure is given by metamodels and execution semantics are given by model transformations.

We made the conscious choice of building our tool on top of cutting-edge Eclipse technologies, as these technologies seem to be the most relevant in the engineering domains our research targets.

One of the most important future directions regarding the tool, would be to redesign it to be a better fit with the Modelverse. [200]. This would mean repositioning the tool as a Modelverse interface, and pushing the execution logic into the Modelverse. Additional engineering tools should be supported in order to make the tool more applicable to industrial-scale problems. The usability of the tool could be improved by additional developments to the user interface, including the modeling facilities and the Eclipse-based productivity functionality. Industrial-scale problems often demand deploying the enacted process onto a server, or nowadays into a cloud service. Our tool is obviously not mature enough in this regard either and effort should be dedicated to this part as well before it gets employed in industrial settings.

# Chapter 8

## Conclusions

This thesis set out to contribute to the state-of-the-art in inconsistency management in Model-based Systems Engineering (MBSE).

We placed our research on the premise that explicit modeling of various aspects of the engineering problem are required to facilitate the efficient, automated reasoning over inconsistencies, their origins, viable resolution strategies and impact on the overall engineering endeavor. This proved to be a very tough challenge and by the end of this research, it became clear, that the explicit modeling of engineering problems is what separates ad-hoc inconsistency management attempts from the systematic, evaluable, improvable and scalable methodologies. This is due to the intangibles in nowadays' systems engineering, such as the tacit and non-encoded domain knowledge, the engineering best practices, rules of thumb, etc.

We very consciously asked for feedback from our industrial partners multiple times throughout this research to understand the thinking and needs of professionals our results could aid in solving their everyday tasks.

As we continuously (incrementally and iteratively) gained a deeper understanding of what inconsistencies are and how they arise, the scope of our research moved towards a higher level of organizational phenomena in engineering settings. Instead of constraining our research to the low-level elements of engineering models (their type system, semantics, etc), we ended up with the conclusion, that managing inconsistencies on the level of the *engineering process* is a missing link in the state-of-the-art, which needs to be addressed. As a consequence, instead of providing one specific way of coping with inconsistencies, we provided a framework for meshing various types of inconsistency management techniques for managing various types of inconsistencies in MBSE settings. Our process-based approach is, however, still very operational, instead being solely descriptive.

Processes popping up as the foundations of a holistic inconsistency management approach, is merely a coincidence though. Processes are the bread and butter of multi-paradigm modeling (MPM). MPM advocates modeling everything explicitly, using the most appropriate formalism(s) on the most appropriate level(s) of abstraction. [202] This advice, however, is not constrained to engineering models only: processes are as vital part of an MPM approach as any other model.

There are two big parts to this thesis: (i) the foundational scientific contributions, discussed in Chapter 4 and Chapter 5; and (ii) the prototype tooling supporting the methodology, discussed in Chapter 7.

## Formalism for modeling engineering endeavors

A formalism for modeling engineering endeavors has been developed. Here, we use the term “endeavor” instead of “process” to emphasize, that it is not only the process that can be modeled using the formalism, but also many other aspects of the engineering work.

The formalism is built on the FTG+PM formalism, which proved to be versatile enough to tailor it to the needs of the stakeholders from the heterogeneous systems engineering community, yet remain semantically valid and sound.

This formalism provides the basis for the inconsistency management methodology discussed in this thesis. It is a standalone contribution in a sense, that it can be reused and reimplemented on other platforms, using other frameworks: the principles will still hold.

The formalism has been extended by multiple important elements, most notably:

- a property model with system characteristics, such as attributes, properties, relationships (Section 5.1.2);
- resource modeling capabilities (Section 5.1.3);
- cost modeling capabilities (Section 5.1.4); and
- capabilities in the FTG (Section 5.4.1).

**Process modeling** Modeling the engineering process at hand is the first step towards modeling the whole engineering setting. The process model (PM) has an explicit type model, the formalism-transformation graph (FTG) charted next to it. This strongly typed process model provides numerous benefits at the later runtime (enactment) phase, but also helps reasoning about multiple metalevels of inconsistency phenomena.

**Modeling system characteristics** Modeling the actual product, that is being engineered, along with the process, was a very important step in our research. System characteristics are vital in terms of expressing inconsistency patterns, their modeling is, however, the most challenging part of our approach. With that said, this is a one-time effort for an engineering process and the efforts are well worth on the long term.

**Resources and costs** Resource and cost modeling elements have been introduced in order to reason about various alternatives of the original process. Since the management of inconsistencies is approached as a process optimization problem, there is an obvious need for quantified performance indicators for each process. We opted for the transit time of the process, provided simulation techniques and tooling for it. Resource modeling is required in order to keep the optimized process models realistic, i.e. constrained by the available human, automated and monetary resources.

## Off-line inconsistency management by process optimization

Having modeled the engineering process and its surroundings enables approaching the task of inconsistency management as a process optimization problem. This state-of-the-art result is a huge step towards holistic approaches which do not only consider the internals of the models, but also the engineering setting the models are situated within. As a consequence, this approach enables reasoning about the impacts of having an inconsistency at some point of the process, and also enables reasoning about the impact of the various inconsistency management techniques on the process. The state-of-the-art is filled with specially tailored inconsistency management techniques, but our work is the first one that delivers a framework to mesh these techniques into an efficient management framework.

**DSE** Design-space exploration of the process, or as we often refer to it: process space exploration, is the means of optimizing the engineering process. Inconsistency management patterns and process performance improvement patterns are captured as model transformations over the explicitly modeled process. By applying these transformations one-by-one, an infinite set of process alternatives can be generated. The DSE approach is a smart search, guided by heuristics, which make sure, the exploration always converges towards an optimum. For that, the newly generated process alternatives are simulated for a chosen performance metric. We chose the transfer time of the process, and presented a state-of-the-art modeling and simulation technique at the end of this research.

## Process enactment

Optimizing processes is already a big step towards managing inconsistencies. But leaving the enactment a completely human effort is a significant threat to the end-result of the engineering endeavor. Because of this, we a process enactment engine has been developed, based on MPM [202] and M@RT[26] principles. The enactment is fully automated, while also supporting human tasks. A process management console is provided to govern the process.

**Service integration** During enactment, the execution of automated activities often requires interaction with engineering tools. Our prototype tool has a demonstrative feature set for Matlab and AMESim. The standard APIs are used to load, manipulate and save models within the engineering tool. This way, the tool also implements a high-level service orchestration machinery. In addition, we elaborated on a state-of-the-art technique on explicitly modeled service interaction at a later stage of this research. The results of that work, however, are not integrated into the tooling.

## On-line inconsistency management

During the enactment phase, inconsistency management is carried out in the background, transparently to the end-user. The characteristics of the engineered system (attributes, properties) are monitored and inconsistent situation are detected in an as-soon-as-possible fashion. Making use of multiple levels of abstraction and multiple types of properties

(ontological vs linguistic), this state-of-the-art technique enables a reasoning mechanism which would not be feasible by human effort at all.

Symbolic mathematics are used in conjunction with the Ecore models to detect inconsistencies. The functionality has been implemented using the Python-based SymPy framework.

## Prototype tooling

During the development of the prototype tooling, we always tried to adhere to the old adage of “one has to eat his own dogfood”. In practical terms, this means shifting software engineering paradigms from writing code towards modeling as much as possible. Overly abstracted models resulting in too many bootstrapping steps, naturally, could lead to a plummeting performance, and therefore, pragmatic trade-offs between modeling and coding have been made. The vast majority (about 75%) of the functionality is, however, modeled.

The tool has been developed as an open-source project, on top of the versatile Eclipse platform, and is available under the EPL1.0 licence. The majority of the functionality has been covered with thorough unit tests in order to make the future improvements and refactorings more efficient.

The tool is available at <https://github.com/david-istvan/icm>.

## 8.1 Evaluation of the research questions

At this point, we reflect on our research questions listed in Section 1.3.

### R1. What are the shortcomings of the state-of-the-art of inconsistency management that hinder the model based engineering of heterogeneous systems?

This question has been addressed with **Contribution 1**, in Section 3.1.

The state-of-the-art of inconsistency management has two main shortcomings. First, the currently available techniques fall short of capturing the *semantic* aspect of models. This is especially concerning in settings where multi-formalism the only way to go about modeling the system, such as the case of engineering heterogeneous systems. Second, currently available techniques focus on the internals of the models, or maybe multiple models, and fall short of understanding higher level issues only understandable on the level of the *process*.

These takeaways make it clear, that the engineering of nowadays’ heterogeneous systems requires new techniques and perhaps technologies to address its issues.

**R2. What is the relation of model (in)consistency to the (in)correctness of the product?**

This question has been addressed with **Contribution 2**, in Chapter 4.

There are no explicit links in the state-of-the-art connecting the notion of consistency between the engineering/business streams of a distributed environment, to the correctness of the final product. We addressed this missing link by positioning model consistency as a *heuristic* to the eventually correct product. By the definition of heuristics, keeping models in a consistent state does not guarantee an eventually correct product, but there are reasons to believe it is useful w.r.t. the eventual correctness. Such reasons include both theoretical and empirical considerations.

This interpretation of inconsistency also bodes well with the premises of inconsistency tolerance. [12]

**R3. What is the most appropriate level of abstraction to approach inconsistency management in model based systems engineering scenarios?**

This question has been addressed with **Contribution 2**, in Section 4.4.

Many state-of-the-art techniques related to this thesis focus on managing inconsistencies on the level of the specific model. Quite frequently, UML is used as a research vehicle to demonstrate how specific types of inconsistencies can be coped with. Such techniques would consider the internals of the models, define techniques in an intimate proximity to the formalism at hand. But while looking too close at the problem, such approaches fail miserably when it comes to real engineering settings of heterogeneous systems. We concluded, that there is a certain need for inconsistency management techniques - complementary to the very low-level and detailed inconsistency management techniques - which approach inconsistency management on a higher level. We concluded that appropriately high level being the *process level*, where traceability with the engineering artifacts is still being maintained, but business-level information (such as resource constraints and scheduling information) can be considered as well.

Choosing processes as the appropriate level of abstraction to approach inconsistency management aligns very well with the efforts previously made in the fields of multi-paradigm modeling [202] and business process modeling [211].

**R4. What formalisms are required to successfully model an engineering endeavor with the intent of identifying inconsistencies across the various models?**

This question has been addressed with **Contributions 3, 6, 7 and 8**, in Section 5.1, Section 5.4, Section 5.5 and Section 5.6, respectively.

Research question **R3**, highlighted the need for approaching inconsistency management on the level of the engineering process, but with traceable engineering activities and artifacts. A *process modeling formalism* is inherently needed for such an approach, but the majority of usable and scalable process/workflow modeling formalisms are positioned for business process modeling. The FTG+PM formalism, on the other hand, fits the bill perfectly when the engineering endeavor is, in fact, model-based. Apart from the modeling the process, *information about the engineered system* has to be also modeled and correlated with the

process model, in order to efficiently identify root causes of inconsistencies. Since the modeled process needs to be modified in order to treat inconsistencies, *resources* and *costs* have to be taken into account as well and supported by modeling formalisms. In the spirit of MPM, the *process modification transformations* are also required to be modeled, which further implies *modeling inconsistencies* themselves. These latter two are sufficiently covered in the state-of-the-art. [165, 177, 178, 184]

#### **R5. How can be the impacts of applying different inconsistency management patterns quantified?**

This question has been addressed with **Contribution 4**, in Section 5.2.

As the old adage goes, “fixing a bug is merely more than replacing a known bug with an unknown one”. Indeed, when applying a management pattern to a specific inconsistency instance, the now restructured process may suffer from new inconsistency instances as well. Additionally: managing an inconsistency may be carried out in various alternative ways, having different performance impacts on the engineering process. In both cases, quantification of the impact of applying an inconsistency management pattern needs to be qualified in order to make decision about applying the pattern or not. We have found that the simulation of processes is a feasible technique, although with a caveat. Every process modeling framework comes with its own, often proprietary and closed-source simulation algorithm. This makes it hard-to-impossible to modify these frameworks to our current needs. Therefore, a general approach has been developed during our research, which would give semantics to processes by DEVS, and use DEVS semantics for quantitative analysis.

#### **R6. Can an inconsistency management technique go beyond being prescriptive and actually enact the inconsistency management techniques chosen for a particular case?**

This question has been addressed with **Contributions 5 and 6**, in Section 5.3 and Section 5.4, respectively.

It is a frequently encountered problem in model-based settings that the models are not leveraged to the maximal effect, e.g. by serving as inputs to conceptualization or early development, but abandoned later on. This is a mistake often made in software engineering and business decision support scenarios. Although a process model for inconsistency management purposes is useful in itself, being able to use that model as the guiding principle during the actual engineering endeavor provides a great added value. Such an employment of the process model is often referred as the *enactment* of the process [35]. Manually *enacting* the process is an error-prone attempt, due to the enormous amount of work it requires, and the manually often undetectable concept drift [183]. Therefore, we provided enactment semantics and implementation to our process models and enabled interfacing with external services in order to further automate the management of inconsistencies in engineering processes.

## 8.2 Future research directions

Hopefully, the results presented in this thesis could serve as basis for future researchers, just as many previous thesis were considered during our research. Out of the plethora of possible future directions, we sketch the ones which are the most probable ones, or are of the highest potential.

**(Partial) process inference** The linchpin of our approach is the modeling of the engineering process and its surroundings. It is a human effort, which is required to be done correctly in order to automate the inconsistency management. Our work does not deal with the automation of the actual process of process modeling. Consequently, inferring the process, at least partially, is the most important future work in our opinion. Interpreting the process as a partially ordered set of activities, the elementary building blocks to process inference would be identifying elementary ordering relations between as many activities as possible. Using domain-specific process templates can be considered as an input to such an inference mechanism as well. But we envision the property model and its variants as the main source of information to the inference. Additionally, business-level information, such as project plans and schedules could be used too. There is a huge potential to be unlocked by minimizing the human factor in the process modeling process. Process mining techniques [189, 193] are prime candidates to build an approach for such a purpose.

**Ontologies as first-class citizens** Somewhat related to the previous point, ontologies [69] can be (and should be) considered as first-class citizens in modern, heterogeneous engineering settings. The two trivial way ontologies could improve our approach are: using ontologies as inputs to the process modeling process, and using them to automatically improve the on-line inconsistency reasoning mechanism along property-capability links (as seen in Section 5.4). Vanherpen [204] investigated ontological reasoning for inconsistency management. A future research direction would preferably pick up both of these works and integrate them.

**Enabling different process modeling languages** We envision an extension to our approach in which arbitrary process modeling languages can be used for the core modeling tasks of inconsistency situations. Processes should be as close to the application domain as possible, as is the case in nowadays process modeling landscape. In addition, however, the semantics of the various process modeling formalisms should at least be correlated with the semantics of inconsistency management [154] shown in this thesis. Being flexible and using the most appropriate process modeling formalism for various domains aligns very well with the core principles of MPM.

**Inconsistency tolerance** Tolerating inconsistencies [12] is something we have briefly investigated during this research (as discussed in Section 5.7.4), but eventually fell outside the narrowing scope of the project. Tolerance is a deep topic and deserves a PhD of its own. This is due to the intangibles of inconsistency management, which we have been trying to tackle throughout these four years, but only that much could be fitted into this time span. The three important types of inconsistency tolerance are parameter

tolerance (acceptance ranges of parameters are relaxed), spatial tolerance (inconsistency rules are relaxed for certain components or sub-components), and temporal tolerance (inconsistencies are tolerated for a certain period of physical or logical time). Out of these, temporal tolerance poses the hardest challenge, but in exchange: promises the most value when tackled. This is due to the fact that tolerance circumvents the shortcomings of overly aggressive inconsistency management strategies and as a consequence, it could tremendously reduce resource usage (time, monetary, human resources). There is some previous work done on the topic in the state-of-the-art, but such a research would ideally start from understanding the deep semantics of model inconsistencies and correlate that with the traces of engineering processes. This would serve as a basis for an appropriate tolerance algebra, stemming from interval algebras [5, 158].

**Improvements to the tool support** Improvements to the tool have been discussed in Chapter 7 in greater details. From a methodological standpoint, the most interesting development directions are (i) integrating the tool with the Modelverse [200]; and (ii) incorporating ontological support into the reasoning logic, as these directions are believed to constitute the foundations of the next generation of engineering techniques and tools [14].

# **Appendices**



## **Appendix A**

### **Metamodels**

This appendix documents the various parts of the process modeling formalism in greater details.

## A.1 Overview

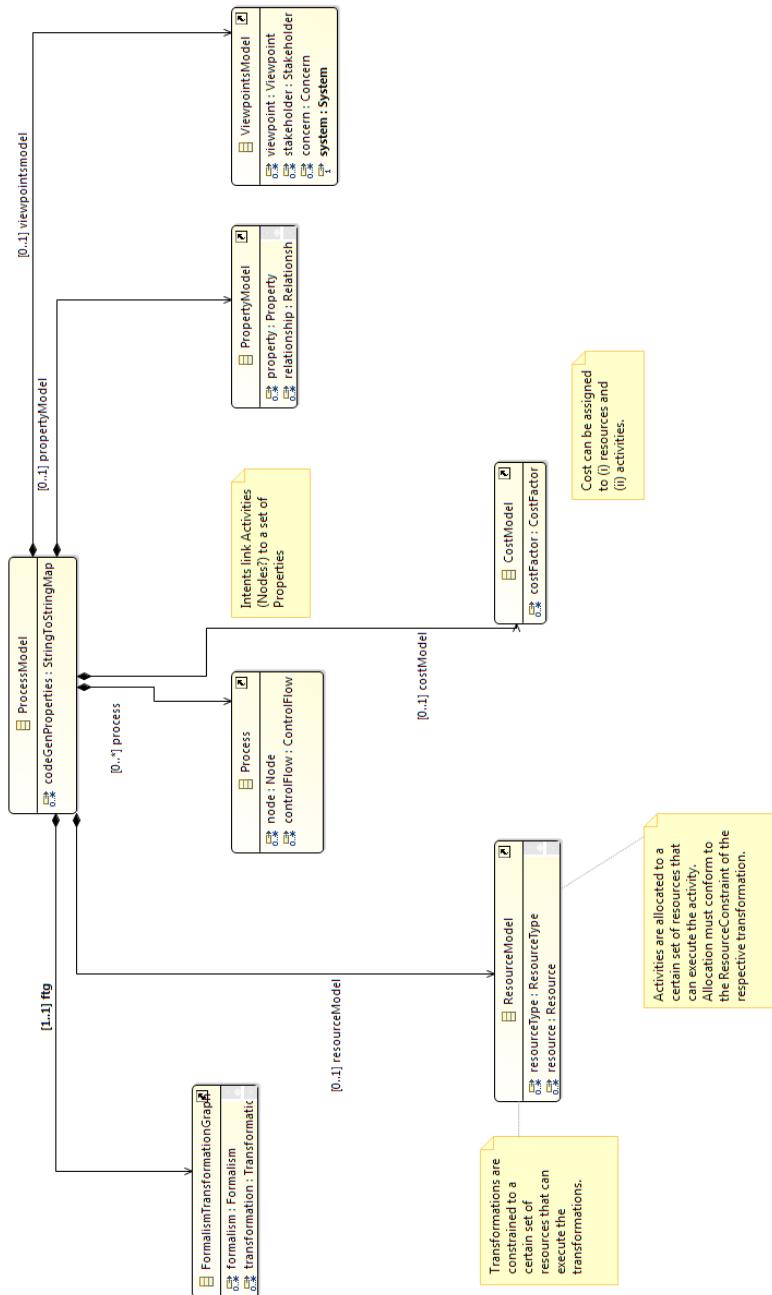


Figure A.1: Top-level overview of the formalism.

## A.2 FTG+PM

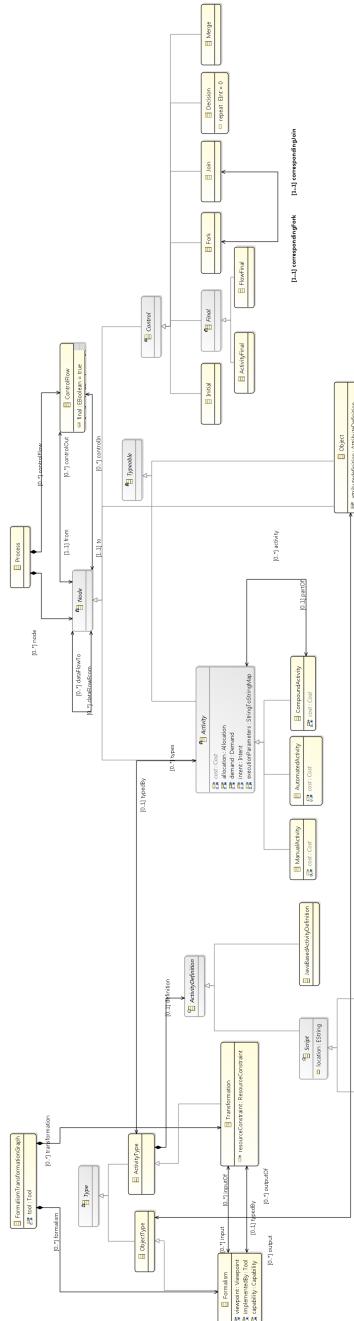


Figure A.2: The FTG+PM modeling part of the formalism.

### A.3 Properties

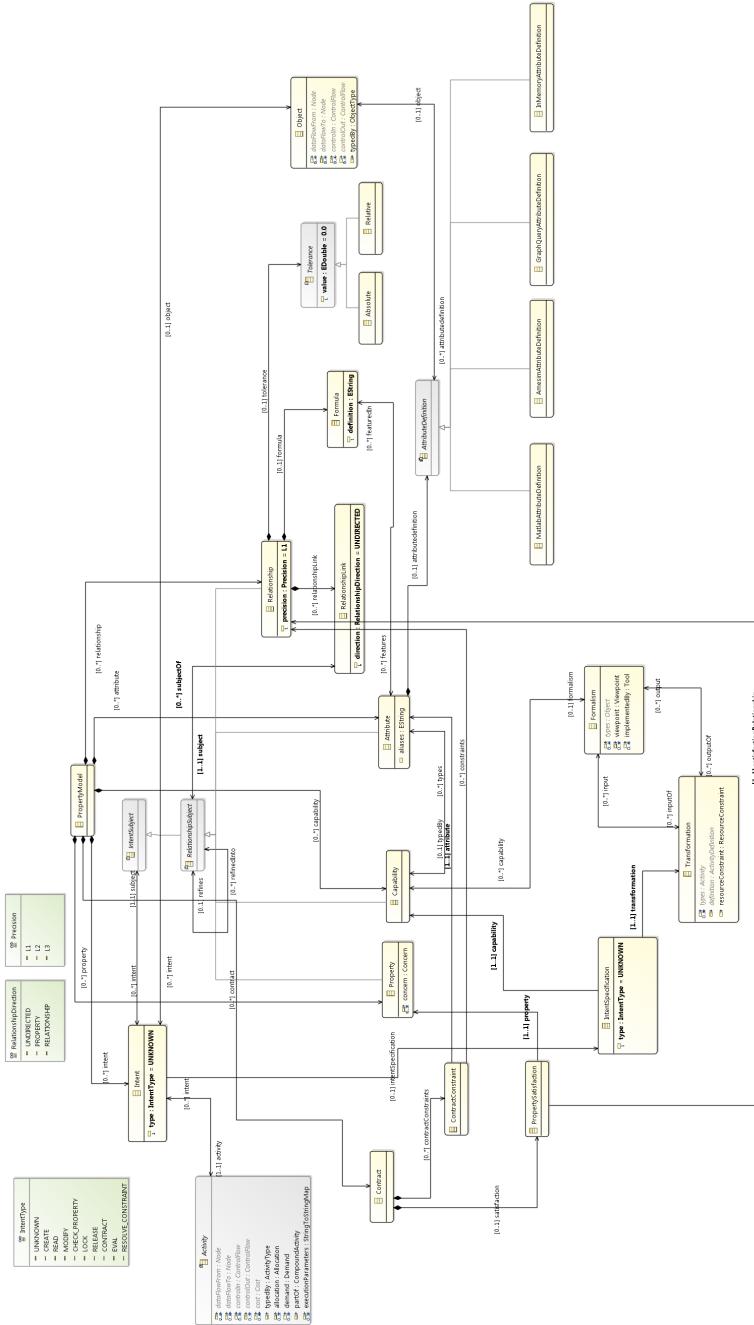


Figure A.3: The property modeling part of the formalism.

## A.4 Resources

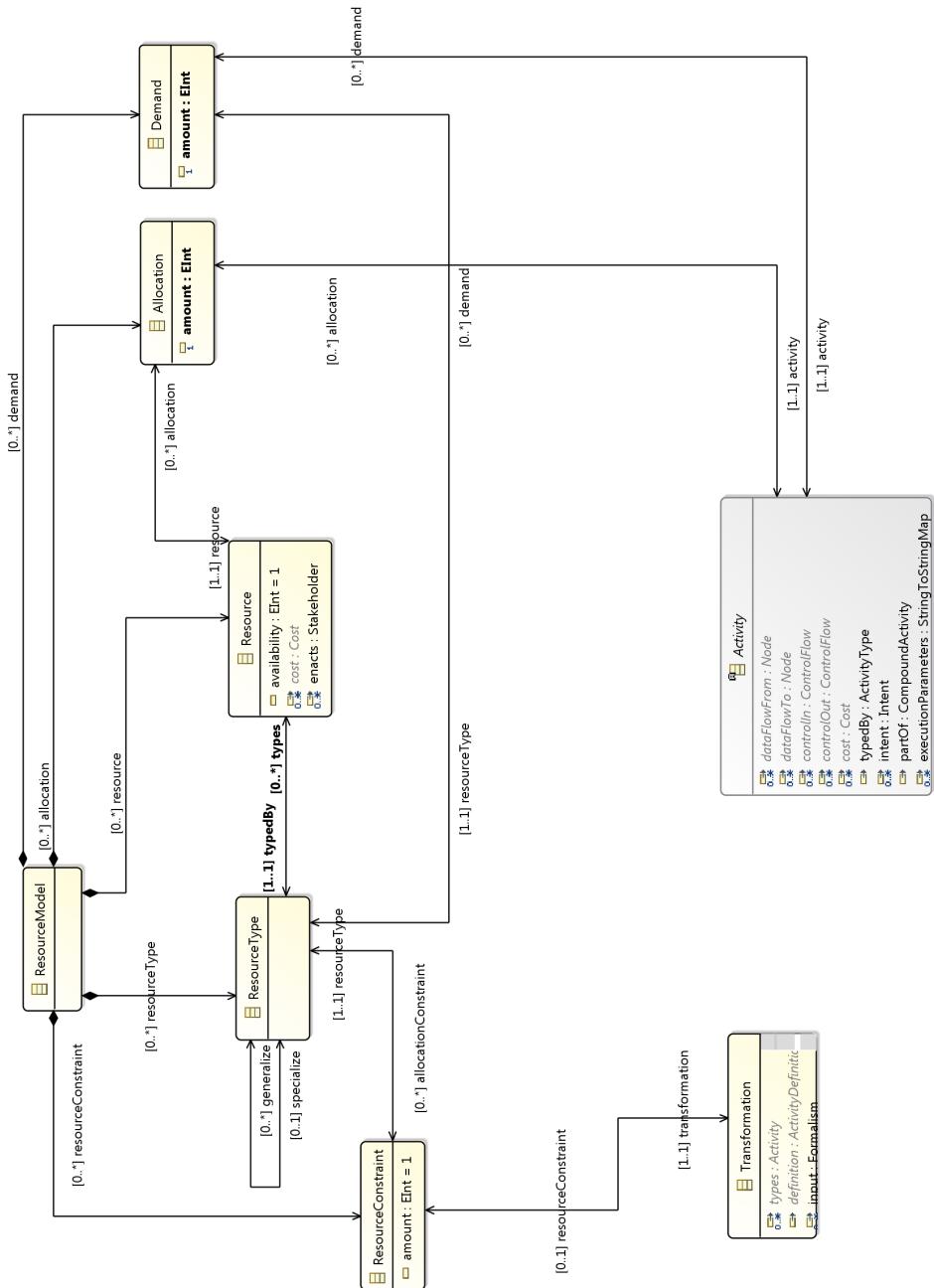


Figure A.4: The resources modeling part of the formalism.

## A.5 Costs

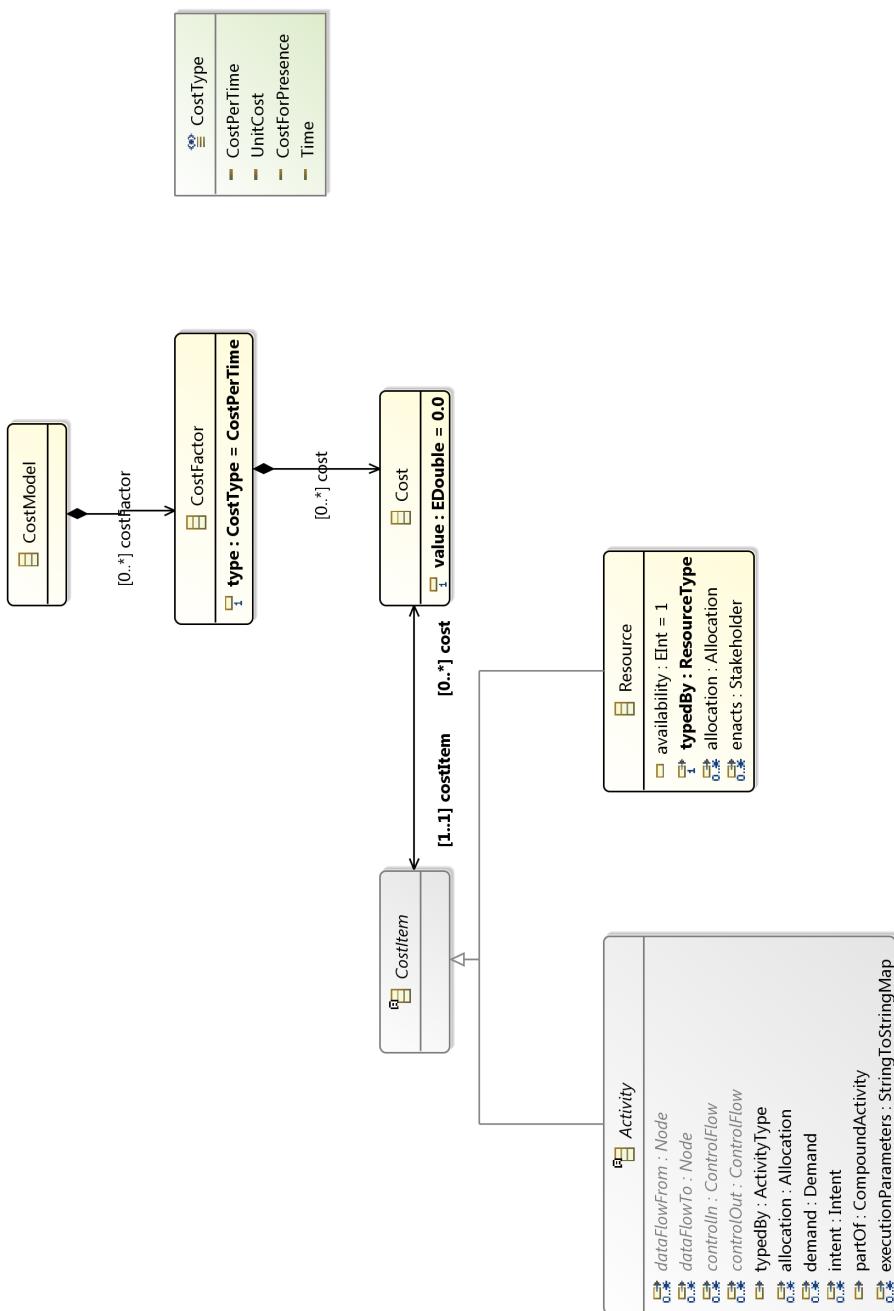


Figure A.5: The cost modeling part of the formalism.

## A.6 Viewpoints

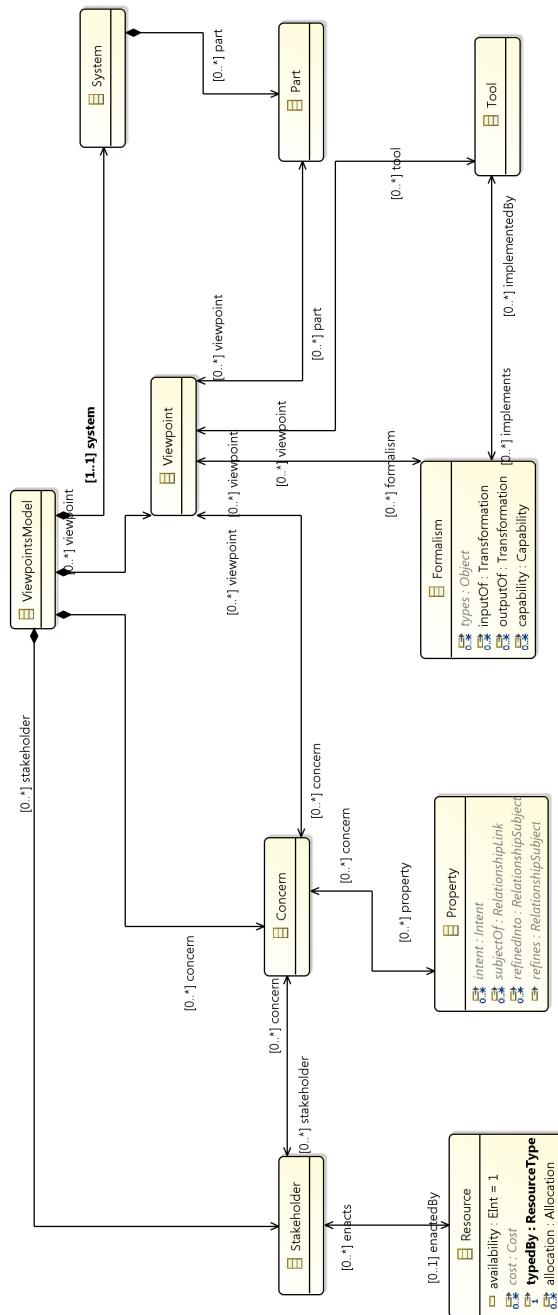


Figure A.6: The viewpoint modeling part of the formalism.

## A.7 Enactment

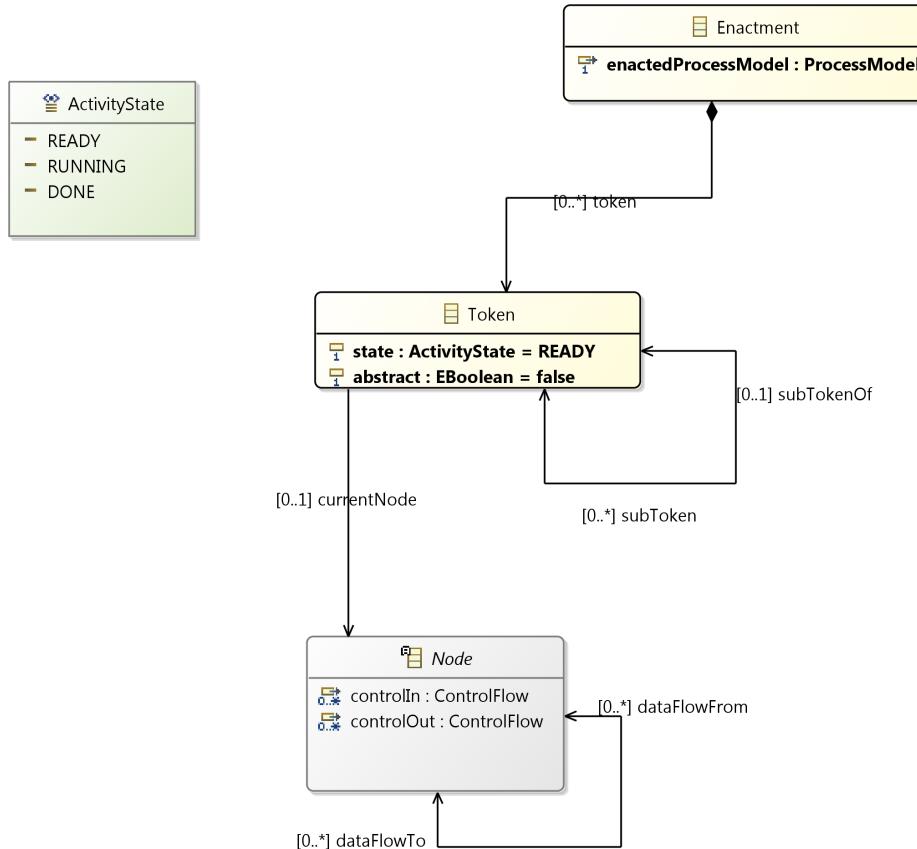


Figure A.7: The enactment modeling part of the formalism.

## **Appendix B**

### **Model queries and transformations**

---

**Listing 5** Patterns of inconsistency.
 

---

```

1 pattern sharedProperty(
2   activity1: Activity, intent1: Intent, property1: Property,
3   activity2: Activity, intent2: Intent, property2: Property){
4     activity1!=activity2;
5     find intent(activity1, property1, intent1);
6     find intent(activity2, property2, intent2);
7     find propertyGloballyReachableFromProperty(property1, property2);
8   }
9
10 pattern sequentialSharedProperty(
11   activity1: Activity, intent1: Intent, property1: Property,
12   activity2: Activity, intent2: Intent, property2: Property){
13   find sharedProperty(
14     activity1, intent1, property1, activity2, intent2, property2);
15   find nodeGloballyReachableFromNode(activity2, activity1);
16 }
17
18 pattern parallelSharedProperty(
19   activity1: Activity, intent1: Intent, property1: Property,
20   activity2: Activity, intent2: Intent, property2: Property,
21   fork: Fork){
22   find sharedProperty(
23     activity1, intent1, property1, activity2, intent2, property2);
24   find parallelActivities2(activity1, activity2, fork,_);
25 }
26
27 pattern readModifySharedProperty(
28   activity1: Activity, property1: Property,
29   activity2: Activity, property2: Property){
30   find sequentialSharedProperty(
31     activity1, intent1, property1, activity2, intent2, property2);
32   Intent.type(intent1, IntentType::READ);
33   Intent.type(intent2, IntentType::MODIFY);
34 }
35
36 pattern modifyModifySharedPropertySequential(
37   activity1: Activity, property1: Property,
38   activity2: Activity, property2: Property){
39   find sequentialSharedProperty(
40     activity1, intent1, property1, activity2, intent2, property2);
41   Intent.type(intent1, IntentType::MODIFY);
42   Intent.type(intent2, IntentType::MODIFY);
43 }
44
45 pattern modifyModifySharedPropertyParallel(
46   activity1: Activity, property1: Property,
47   activity2: Activity, property2: Property,
48   fork: Fork){
49   find parallelSharedProperty(
50     activity1, intent1, property1, activity2, intent2, property2, fork);
51   Intent.type(intent1, IntentType::MODIFY);
52   Intent.type(intent2, IntentType::MODIFY);
53 }
```

---

---

**Listing 6** Patterns resource allocation.
 

---

```

1 pattern allocatedActivity(
2   activity: Activity, allocation: Allocation, resource: Resource){
3   neg find compoundActivity(activity);
4   Activity.allocation(activity, allocation);
5   Allocation.resource(allocation, resource);
6 }
7
8 pattern unAllocatedActivity(activity: Activity){
9   neg find compoundActivity(activity);
10  neg find allocatedActivity(activity, _, _);
11 }
12
13 pattern demand(activity: Activity, demand: Demand, resourceType: ResourceType){
14   neg find compoundActivity(activity);
15   Activity.demand(activity, demand);
16   Demand.resourceType(demand, resourceType);
17 }
18
19 pattern resourceInstance(resource: Resource, resourceType: ResourceType){
20   Resource.typedBy(resource, resourceType);
21 }
22
23 pattern unsatisfiedDemand(activity: Activity){
24   neg find compoundActivity(activity);
25
26   find demand(activity, demand, resourceType);
27   find allocatedActivity(activity, allocation, resource);
28
29   find resourceInstance(resource, resourceType);
30
31   Demand.amount(demand, dAmount);
32   Allocation.amount(allocation, aAmount);
33
34   check(aAmount < dAmount);
35 }
36
37 pattern satisfiedDemand(activity: Activity){
38   neg find compoundActivity(activity);
39   neg find unsatisfiedDemand(activity);
40 }
41
42 pattern satisfiableDemand(activity: Activity, resourceType: ResourceType){
43   neg find compoundActivity(activity);
44
45   find demand(activity, demand, resourceType);
46
47   find resourceInstance(resource, resourceType);
48
49   Demand.amount(demand, dAmount);
50   Resource.availability(resource, availability);
51
52   check(availability >= dAmount);
53 }
54
55 pattern unsatisfiableDemand(activity: Activity, resourceType: ResourceType){
56   neg find compoundActivity(activity);
57
58   find demand(activity, demand, resourceType);
59
60   find resourceInstance(resource, resourceType);
61
62   Demand.amount(demand, dAmount);
63   Resource.availability(resource, availability);
64
65   check(availability < dAmount);
66 }
```

---

**Listing 7** Management transformation rules for the sequential read-modify inconsistency pattern. Reuses patterns from Listing 5.

```

1  /**
2  * Reordering
3 */
4 val readModifyReorder = new DSETransformationRule(
5     unmanagedReadModify,
6     new UnmanagedReadModifyProcessor() {
7         override process(Activity activity1, Property property1,
8             Activity activity2, Property property2) {
9                 val tmp = createManualActivity("tmp");
10
11                 tmp.controlIn.addAll(activity1.controlIn)
12                 activity1.controlIn.removeAll(tmp.controlIn)
13
14                 tmp.controlOut.addAll(activity1.controlOut)
15                 activity1.controlOut.removeAll(tmp.controlOut)
16
17                 activity1.controlIn.addAll(activity2.controlIn)
18                 activity2.controlIn.removeAll(activity1.controlIn)
19
20                 activity1.controlOut.addAll(activity2.controlOut)
21                 activity2.controlOut.removeAll(activity1.controlOut)
22
23                 activity2.controlIn.addAll(tmp.controlIn)
24                 tmp.controlIn.removeAll(activity2.controlIn)
25
26                 activity2.controlOut.addAll(tmp.controlOut)
27                 tmp.controlOut.removeAll(activity2.controlOut)
28             }
29     }
30 )
31
32 /**
33 * Check property
34 */
35 val readModifyAugmentWithCheck = new DSETransformationRule(
36     unmanagedReadModify2,
37     new UnmanagedReadModify2Processor() {
38         override process(Activity activity1, Property property1,
39             Activity activity2, Property property2) {
40                 createDecision(activity2, property1, activity1)
41             }
42     }
43 )
44
45 val readModifyAugmentWithContract = new DSETransformationRule(
46     unmanagedReadModify3,
47     new UnmanagedReadModify3Processor() {
48         override process(Activity activity1, Property property1,
49             Activity activity2, Property property2) {
50                 createContract(activity1, #[property1], activity1)
51             }
52     }
53 )
54
55 val modifyModifyAugmentWithContract = new DSETransformationRule(
56     unmanagedModifySequential,
57     new UnmanagedModifySequentialProcessor() {
58         override process(Activity activity1, Property property1,
59             Activity activity2, Property property2) {
60                 createContract(activity1, #[property1], activity1)
61             }
62     }
63 )

```

---

**Listing 8** Transformation rule of the contract-based management pattern for the for the parallel modify-modify inconsistency pattern. Reuses patterns from Listing 5.

---

```

1 val addContract = new DSETtransformationRule(
2   unmanagedModifyModifyParallel,
3   new UnmanagedModifyModifyParallelProcessor() {
4     override process(Activity activity1, Property property1,
5       Activity activity2, Property property2, Fork fork) {
6       createContract(fork, #[property1, property2], activity1)
7     }
8   }
9 )

```

---

**Listing 9** Soft and hard objectives for the process space exploration.

---

```

1 val consistencyObjective =
2   new ConstraintsObjective()
3   .withSoftConstraint("consistencyObjective", unmanagedReadModify, 10d)
4   .withComparator(Comparators.LOWER_IS_BETTER)
5
6 val cheapestProcessObjective =
7   new CheapestProcessSoftObjective()
8   .withComparator(Comparators.LOWER_IS_BETTER)
9
10 val allocationObjectives =
11   new ConstraintsObjective("validAllocation")
12   .withHardConstraint(unAllocatedActivity, ModelQueryType::NO_MATCH)
13   .withLevel(0)
14
15 val validationObjectives =
16   new ConstraintsObjective("validProcess")
17   .withHardConstraint(
18     initNodeWithInvalidNumberOfControlOut, ModelQueryType::NO_MATCH)
19   .withHardConstraint(
20     initNodeWithControlIn, ModelQueryType::NO_MATCH)
21   .withHardConstraint(
22     finalNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
23   .withHardConstraint(
24     finalNodeWithControlOut, ModelQueryType::NO_MATCH)
25   .withHardConstraint(
26     forkNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
27   .withHardConstraint(
28     forkNodeWithInvalidNumberOfOuts, ModelQueryType::NO_MATCH)
29   .withHardConstraint(
30     joinNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
31   .withHardConstraint(
32     joinNodeWithInvalidNumberOfOuts, ModelQueryType::NO_MATCH)
33   .withHardConstraint(
34     decisionNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
35   .withHardConstraint(
36     decisionNodeWithInvalidNumberOfOuts, ModelQueryType::NO_MATCH)
37   .withHardConstraint(
38     activityWithInvalidNumberOfControlIn, ModelQueryType::NO_MATCH)
39   .withHardConstraint(
40     activityWithInvalidNumberOfControlOut, ModelQueryType::NO_MATCH)
41   .withHardConstraint(
42     controlFlowWithInvalidNumberOfControlFrom, ModelQueryType::NO_MATCH)
43   .withHardConstraint(
44     controlFlowWithInvalidNumberOfControlTo, ModelQueryType::NO_MATCH)
45   .withHardConstraint(
46     redundantControlFlows, ModelQueryType::NO_MATCH)
47   .withHardConstraint(
48     finalNotReachableFromNode, ModelQueryType::NO_MATCH)
49   .withHardConstraint(
50     initDoesNotReachNode, ModelQueryType::NO_MATCH)
51   .withLevel(0)

```

---

**Listing 10** Model patterns used in the process enactment runtime - Part 1.

```

1 pattern activeToken(token: Token) {
2   Token.abstract(token, a);
3   a == false;
4 }
5
6 pattern tokenInNode(token : Token, node : Node){
7   find activeToken(token);
8   Token.currentNode(token, node);
9 }
10
11 pattern tokenInActivity(token : Token, node : Activity){
12   find activeToken(token);
13   find tokenInNode(token, node);
14 }
15
16 pattern activity(node: Node){
17   Activity(node);
18 }
19
20 pattern attributeModificationActivity(
21   activity: Activity, attribute: Attribute){
22   Intent.activity(intent, activity);
23   Intent.subject(intent, attribute);
24   Intent.type(intent, ::MODIFY);
25 }
26
27 pattern fireable(
28   token: Token, controlFlow: ControlFlow, fromNode: Node, toNode: Node) {
29   //fire from a non-activity
30   find tokenInNode(token, fromNode);
31   ControlFlow.from(controlFlow, fromNode);
32   ControlFlow.to(controlFlow, toNode);
33   neg find activity(fromNode);
34   neg find splitTokenInJoin(token, fromNode);
35 }
36 or{
37   //fire from a non-activity to a node where
38   //the direct sibling of the token is located
39   find tokenInNode(token, fromNode);
40   ControlFlow.from(controlFlow, fromNode);
41   ControlFlow.to(controlFlow, toNode);
42   neg find activity(fromNode);
43   find splitTokenInJoin(token, fromNode);
44   find splitTokenInJoin(token2, toNode);
45   find directSiblingTokens(token, token2);
46 }
47 or{
48   //fire from an activity
49   find tokenInNode(token, fromNode);
50   ControlFlow.from(controlFlow, fromNode);
51   ControlFlow.to(controlFlow, toNode);
52   find doneActivity(token, fromNode);
53 }
54
55 pattern availableActivity(
56   token: Token, controlFlow: ControlFlow, activity: Activity){
57   find fireable(token, controlFlow, _, activity);
58 }
59
60 pattern availableFinish(token: Token, controlFlow: ControlFlow, final: Final){
61   find fireable(token, controlFlow, _, final);
62 }
```

---

**Listing 11** Model patterns used in the process enactment runtime - Part 2. (Continuation of Listing 10.)

---

```

1 pattern fireableToControl(token: Token, fromNode: Node, control: Control) {
2   find fireable(token, _, fromNode, control);
3   Fork(control);
4 }or
5   find fireable(token, _, fromNode, control);
6   Join(control);
7 }or
8   find fireable(token, _, fromNode, control);
9   Decision(control);
10 }or
11   find fireable(token, _, fromNode, control);
12   Merge(control);
13 }
14
15 pattern splitTokenInJoin(token: Token, join: Join){
16   find tokenInNode(token, join);
17   find siblingTokens(token, token2);
18   find tokensInDifferentNodes(token, token2);
19 }
20
21 pattern joinable(join: Join){
22   find tokenInNode(token, join);
23   find siblingTokens(token, token2);
24   neg find tokensInDifferentNodes(token, token2);
25 }
26
27 pattern forkable(fork: Fork, token: Token){
28   find tokenInNode(token, fork);
29 }
30
31 pattern subTokenOf(subToken: Token, parent: Token){
32   Token.subTokenOf(subToken, parent);
33 }
34
35 pattern siblingTokens(token1: Token, token2: Token){
36   find activeToken(token1);
37   find activeToken(token2);
38   find subTokenOf+(token1, parent);
39   find subTokenOf+(token2, parent);
40   token1 != token2;
41 }
42
43 pattern directSiblingTokens(token1: Token, token2: Token){
44   find activeToken(token1);
45   find activeToken(token2);
46   find subTokenOf(token1, parent);
47   find subTokenOf(token2, parent);
48   token1 != token2;
49 }
50
51 pattern tokensInDifferentNodes(token1: Token, token2: Token){
52   Token.currentNode(token1, node1);
53   Token.currentNode(token2, node2);
54   node1 != node2;
55 }
56
57 //to detect runnable activity
58 pattern readyActivity(token : Token, node : Activity){
59   find tokenInActivity(token, node);
60   Token.state(token, ::READY);
61 }
```

---

**Listing 12** Model patterns used in the process enactment runtime - Part 3. (Continuation of Listing 11.)

```

1 //to detect running activity
2 pattern runningActivity(token : Token, node : Activity){
3     find tokenInActivity(token, node);
4     Token.state(token, ::RUNNING);
5 }
6
7 //to detect executed activity
8 pattern doneActivity(token : Token, node : Activity){
9     find tokenInActivity(token, node);
10    Token.state(token, ::DONE);
11 }
12
13 //to split tokens
14 pattern tokenInFork(token : Token, node : Fork){
15     find tokenInNode(token, node);
16 }
17
18 //to re-join tokens
19 pattern tokenInJoin(token : Token, node : Join){
20     find tokenInNode(token, node);
21 }
22
23 //to multiplex tokens
24 pattern tokenInDecision(token : Token, node : Decision){
25     find tokenInNode(token, node);
26 }
27
28 //to de-multiplex tokens
29 pattern tokenInMerge(token : Token, node : Merge){
30     find tokenInNode(token, node);
31 }
32
33 //to designate the end of an execution
34 pattern finishedProcess(token : Token, node : FlowFinal){
35     find tokenInNode(token, node);
36 }
37
38 //to enable moving tokens from one node to another
39 pattern enabledTransition(fromNode : Node, toNode : Node, token: Token){
40     ControlFlow.from(controlFlow, fromNode);
41     ControlFlow.to(controlFlow, toNode);
42     find tokenInNode(token, fromNode);
43     Token.state(token, ::DONE);
44 }
```

**Listing 13** Elementary model transformations used in the process enactment runtime.  
Reuses the patterns from Listing 10, Listing 11 and and Listing 12.

```

1 val fireToControlRule = createRule.name("fire to control")
2   .precondition(fireableToControl)
3   .action [
4     logger.debug(String.format(
5       "Firing token %s from node %s to control node %s.",
6       token, fromNode, control))
7     token.currentNode = control
8   ]
9   .build
10
11 val forkableRule = createRule.name("forkable")
12   .precondition(forkable)
13   .action [
14     logger.debug(String.format("Forking token %s at %s.", token, fork))
15     // de-activate parent
16     token.abstract = true
17
18     // create sub-tokens
19     for (ctrlOut : fork.controlOut) {
20       val newToken = EnactmentFactory.eINSTANCE.createToken
21       newToken.subTokenOf = token
22       enactment.token.add(newToken)
23       newToken.currentNode = ctrlOut.to
24       if (ctrlOut.to instanceof Activity) {
25         newToken.state = ActivityState::READY
26       }
27     }
28     println("forkable")
29   ]
30   .build
31
32 val joinableRule = createRule.name("joinable")
33   .precondition(joinable)
34   .action [
35     logger.debug(String.format("Joining tokens at %s.", join))
36     val tokenMatches = queryEngine
37       .getMatcher(TokenInNodeQuerySpecification.instance)
38       .allMatches.filter [ match | match.node.equals(join)
39     ] // each token at this point should be joinable
40     logger
41     .debug(String.format("Joinable tokens: %s.",
42       tokenMatches.map[tm|tm.token].toList))
43
44     // activate parent
45     val parentToken = tokenMatches.head.token.subTokenOf
46     parentToken.abstract = false
47     parentToken.currentNode = join
48
49     // remove subs
50     for (tokenMatch : tokenMatches) {
51       enactment.token.remove(tokenMatch.token)
52     }
53     println("joinable")
54   ].build
```



## Appendix C

# DEVS library for processes

This appendix documents full library of atomic DEVS models for processes. The library has been developed by Yentl Van Tendeloo.

---

**Listing 14** Atomic DEVS model library for processes - Part 1.

---

```
1 from pypdevs.DEVS import *
2
3 class ResourceHandler(AtomicDEVS):
4     def __init__(self, resources):
5         AtomicDEVS.__init__(self, "__resource_handler")
6         self.state = {'resources': resources, 'queue': []}
7         self.elapsed = 0.0
8         self.resource_in = self.addInPort("resource_in")
9         self.resource_out = self.addOutPort("resource_out")
10
11     def extTransition(self, inputs):
12         for inp in inputs[self.resource_in]:
13             if inp['type'] == "request":
14                 # Queue the request
15                 self.state['queue'].append(inp['id'])
16             elif inp['type'] == "release":
17                 # Increment the number of available resources
18                 self.state['resources'] += 1
19
20         return self.state
21
22     def intTransition(self):
23         # Processed a request that could be processed
24         self.state['resources'] -= 1
25         del self.state['queue'][0]
26
27         return self.state
28
29     def outputFnc(self):
30         return {self.resource_out: {'id': self.state['queue'][0]}}
31
32     def timeAdvance(self):
33         if self.state['queue'] and self.state['resources']:
34             # Can grant a resource
35             return 0.0
36         else:
37             # No request queued, or no available resources to handle the request
38             return float('inf')
```

---

**Listing 15** Atomic DEVS model library for processes - Part 2.

```

1  class ActivityState(object):
2      def __init__(self, name, distribution):
3          self.timer = float('inf')
4          self.mode = "inactive"
5          self.counter = 0
6          self.name = name
7          self.distribution = distribution
8
9      def __str__(self):
10         return str(vars(self))
11
12     def random_sample(self):
13         return self.distribution(self.counter)
14
15 class Activity(AtomicDEVS):
16     def __init__(self, name, distribution):
17         AtomicDEVS.__init__(self, name)
18         self.state = ActivityState(name, distribution)
19         self.elapsed = 0.0
20         self.control_in = self.addInPort("control_in")
21         self.resource_in = self.addInPort("resource_in")
22         self.control_out = self.addOutPort("control_out")
23         self.resource_out = self.addOutPort("resource_out")
24
25     def intTransition(self):
26         self.state.timer -= self.timeAdvance()
27         if self.state.mode == "request_resource":
28             # Go and request the required resource
29             self.state.mode = "wait_resource"
30             self.state.timer = float('inf')
31         elif self.state.mode == "active":
32             # Finished execution, so release resources
33             self.state.mode = "inactive"
34             self.state.timer = 0.0
35             self.state.timer = float('inf')
36             self.state.counter += 1
37         return self.state
38
39     def extTransition(self, inputs):
40         self.state.timer -= self.elapsed
41         if self.state.mode == "inactive" and self.control_in in inputs:
42             # Got control token, so ask for the required resources
43             self.state.mode = "request_resource"
44             self.state.timer = 0.0
45             # NOTE this violates DEVS, though is easy to debug
46             #print("Activate " + str(self.state.name) + " at time " + str(self.
47             #    ↪ time_last[0] + self.elapsed))
48         elif self.state.mode == "wait_resource" and self.resource_in in inputs and
49             #    ↪ inputs[self.resource_in]['id'] == "%s-%s" % (self.state.name, self.
50             #    ↪ state.counter):
51             # Got required resources, so start execution
52             self.state.mode = "active"
53             self.state.timer = self.state.random_sample()
54         return self.state
55
56     def timeAdvance(self):
57         return self.state.timer
58
59     def outputFnc(self):
60         if self.state.mode == "active":
61             # Output the control token to the next model in line, and release the
62             #    ↪ resources
63             return {self.control_out: {}, self.resource_out: [{"type": "release"}]}
64         elif self.state.mode == "request_resource":
65             # Output a request for resources with a specified ID (used to find out
66             #    ↪ whether this was our request)
67             return {self.resource_out: [{"type": "request", 'id': "%s-%s" % (self.
68             #    ↪ state.name, self.state.counter)}]}
69         else:
70             return {}

```

---

**Listing 16** Atomic DEVS model library for processes - Part 3.
 

---

```

1  class ParallelSplit(AtomicDEVS):
2      def __init__(self, name):
3          AtomicDEVS.__init__(self, name)
4          self.control_in = self.addInPort("control_in")
5          self.control_out = self.addOutPort("control_out")
6          self.state = False
7
8      def intTransition(self):
9          return False
10
11     def extTransition(self, inputs):
12         return True
13
14     def outputFnc(self):
15         return {self.control_out: {}}
16
17     def timeAdvance(self):
18         if self.state:
19             return 0.0
20         else:
21             return float('inf')
22
23 class Synchronization(AtomicDEVS):
24     def __init__(self, name, counter):
25         AtomicDEVS.__init__(self, name)
26         self.state = {'current': counter, 'max': counter}
27         self.control_in = self.addInPort("control_in")
28         self.control_out = self.addOutPort("control_out")
29
30     def intTransition(self):
31         self.state['current'] = self.state['max']
32         return self.state
33
34     def extTransition(self, inputs):
35         self.state['current'] -= 1
36         return self.state
37
38     def timeAdvance(self):
39         if self.state['current'] == 0:
40             return 0.0
41         else:
42             return float('inf')
43
44     def outputFnc(self):
45         return {self.control_out: {}}
46
47 class ExclusiveChoiceState(object):
48     def __init__(self, outputs, distribution):
49         self.counter = 0
50         self.outputs = outputs
51         self.choice = None
52         self.distribution = distribution
53
54     def __str__(self):
55         return str(vars(self))
56
57     def make_choice(self):
58         return self.distribution(self.counter)

```

---

---

**Listing 17** Atomic DEVS model library for processes - Part 4.

---

```

1 class ExclusiveChoice(AtomicDEVS):
2     def __init__(self, name, outputs, distribution):
3         AtomicDEVS.__init__(self, name)
4         self.state = ExclusiveChoiceState(outputs, distribution)
5         self.control_in = self.addInPort("control_in")
6         self.control_out = [self.addOutPort("control_out_%s" % i) for i in range(
7             → outputs)]
8
9     def intTransition(self):
10        self.state.choice = None
11        self.state.counter += 1
12        return self.state
13
14     def extTransition(self, inputs):
15        # Got a control token, so have to make a choice
16        self.state.choice = self.state.make_choice()
17        return self.state
18
19     def outputFnc(self):
20        return {self.control_out[self.state.choice]: {}}
21
22     def timeAdvance(self):
23        if self.state.choice is not None:
24            return 0.0
25        else:
26            return float('inf')
27
28 class SimpleMerge(AtomicDEVS):
29     def __init__(self, name):
30         AtomicDEVS.__init__(self, name)
31         self.state = False
32         self.control_in = self.addInPort("control_in")
33         self.control_out = self.addOutPort("control_out")
34
35     def intTransition(self):
36        return False
37
38     def extTransition(self, inputs):
39        return True
40
41     def outputFnc(self):
42        return {self.control_out: {}}
43
44     def timeAdvance(self):
45        if self.state:
46            return 0.0
47        else:
48            return float('inf')
49
50 class MultiInstanceState(object):
51     def __init__(self, name, num, distribution):
52         self.spawned = num
53         self.collected = 0
54         self.counter = 0
55         self.mode = "inactive"
56         self.running_tasks = []
57         self.requested = []
58         self.name = name
59         self.distribution = distribution
60
61     def __str__(self):
62        return str(vars(self))
63
64     def task_time(self):
65        return self.distribution(self.counter)

```

---

---

**Listing 18** Atomic DEVS model library for processes - Part 5.
 

---

```

1 class MultiInstance(AtomicDEVS):
2     def __init__(self, name, num, distribution):
3         AtomicDEVS.__init__(self, name)
4         self.state = MultiInstanceState(name, num, distribution)
5         self.control_in = self.addInPort("control_in")
6         self.resource_in = self.addInPort("resource_in")
7         self.control_out = self.addOutPort("control_out")
8         self.resource_out = self.addOutPort("resource_out")
9
10    def intTransition(self):
11        ta = self.timeAdvance()
12        for t in self.state.running_tasks:
13            t[0] -= ta
14
15        if self.state.mode == "active":
16            # Finished an instance, so pop it
17            del self.state.running_tasks[0]
18            self.state.collected += 1
19            if self.state.collected == self.state.spawned:
20                self.state.mode = "finish"
21
22        elif self.state.mode == "request_resources":
23            # Requested resources, so be ready for a response
24            self.state.mode = "active"
25
26        elif self.state.mode == "finish":
27            self.state.mode = "inactive"
28            self.state.collected = 0
29            self.state.counter += 1
30            self.state.running_tasks = []
31
32        return self.state
33
34    def extTransition(self, inputs):
35        # Got input, so have to spawn #num of them
36        for t in self.state.running_tasks:
37            t[0] -= self.elapsed
38
39        if self.state.mode == "inactive" and self.control_in in inputs:
40            self.state.mode = "request_resources"
41            self.state.requested = ["%s-%s-%s" % (self.state.name, self.state.counter
42                                     ↔, i) for i in range(self.state.spawned)]
43
44        if self.state.mode in ["active", "release_resource"] and self.resource_in in
45            ↔ inputs:
46            # Got a resource, so allocate it to an activity
47            self.state.running_tasks.append([self.state.task_time(), self.state.
48                                             ↔ requested.pop(0)])
49            # NOTE this violates DEVS, though is easy to debug
50            print("Spawn " + str(self.state.running_tasks[-1][1]) + " at time " +
51                  ↔ str(self.time_last[0] + self.elapsed))
52            self.state.running_tasks.sort()
53
54        return self.state
55
56    def outputFnc(self):
57        if self.state.mode == "request_resources":
58            # Request all resources in one go
59            return {self.resource_out: [{"type": 'request', 'id': i} for i in self.
60                                     ↔ state.requested]}
61
62        elif self.state.mode == "active":
63            # Finished an instance, so release it
64            return {self.resource_out: [{"type": 'release'}]}
65
66        elif self.state.mode == "finish":
67            # Finished execution of all, so pass on token
68            return {self.control_out: {}}
69
70        else:
71            return {}
72
73    def timeAdvance(self):
74        if self.state.mode == "finish":
75            return 0.0
76
77        elif self.state.running_tasks:
78            return self.state.running_tasks[0][0]
79
80        elif self.state.mode == "request_resources":
81            return 0.0
82
83        else:
84            return float('inf')

```

---

---

**Listing 19** Atomic DEVS model library for processes - Part 6.

---

```
1 class Initial(AtomicDEVS):
2     def __init__(self):
3         AtomicDEVS.__init__(self, "Initial")
4         self.control_out = self.addOutPort("control_out")
5         self.state = True
6
7     def intTransition(self):
8         return False
9
10    def outputFnc(self):
11        return {self.control_out: {}}
12
13    def timeAdvance(self):
14        if self.state:
15            return 0.0
16        else:
17            return float('inf')
18
19 class Finish(AtomicDEVS):
20     def __init__(self):
21         AtomicDEVS.__init__(self, "Finish")
22         self.control_in = self.addInPort("control_out")
23         self.state = None
24
25     def extTransition(self, inputs):
26         return self.elapsed
27
28     def timeAdvance(self):
29         return float('inf')
```

---

---

**Listing 20** Coupled DEVS model of the running example - Part 1.
 

---

```

1 from library import *
2 from pypdevs.simulator import Simulator
3 from pypdevs.DEVS import CoupledDEVS
4 import random
5 import math
6
7 def df(iteration):
8     result = random.random() < {0: 0.99, 1: 0.9, 2: 0.8, 3: 0.5, 4: 0.2, 5: 0.1, 6:
9         ↪ 0.02}.get(iteration, 0.0)
10    return int(result)
11
12 def cf(mean):
13     def cf_int(iteration):
14         mu = mean * (1-math.exp(-(iteration+1) / 0.7))
15         sigma = mu*0.15625
16         return max(0.0, random.gauss(mu, sigma))
17     return cf_int
18
19 class Example(CoupledDEVS):
20     def __init__(self, nresources):
21         CoupledDEVS.__init__(self, "example")
22         initial = self.addSubModel(Initial())
23         req = self.addSubModel(Activity("define_requirements", cf(10.0)))
24         merge = self.addSubModel(SimpleMerge("merge"))
25         do_model = self.addSubModel(Activity("update_model", cf(100.0)))
26         split = self.addSubModel(ParallelSplit("split"))
27         sim = self.addSubModel(MultiInstance("simulation", 10, cf(20.0)))
28         check = self.addSubModel(Activity("checking", cf(30.0)))
29         synchronization = self.addSubModel(Synchronization("synchronization", 2))
30         evaluation = self.addSubModel(Activity("evaluation", cf(0.1)))
31         choice = self.addSubModel(ExclusiveChoice("choice", 2, df))
32         self.finish = self.addSubModel(Finish())
33
34         resources = self.addSubModel(ResourceHandler(nresources))
35
36         self.connectPorts(initial.control_out, req.control_in)
37         self.connectPorts(req.control_out, merge.control_in)
38         self.connectPorts(choice.control_out[1], merge.control_in)
39         self.connectPorts(merge.control_out, do_model.control_in)
40         self.connectPorts(do_model.control_out, split.control_in)
41         self.connectPorts(split.control_out, sim.control_in)
42         self.connectPorts(split.control_out, check.control_in)
43         self.connectPorts(sim.control_out, synchronization.control_in)
44         self.connectPorts(check.control_out, synchronization.control_in)
45         self.connectPorts(synchronization.control_out, evaluation.control_in)
46         self.connectPorts(evaluation.control_out, choice.control_in)
47         self.connectPorts(choice.control_out[0], self.finish.control_in)
48
49         self.connectPorts(req.resource_out, resources.resource_in)
50         self.connectPorts(resources.resource_out, req.resource_in)
51         self.connectPorts(do_model.resource_out, resources.resource_in)
52         self.connectPorts(resources.resource_out, do_model.resource_in)
53         self.connectPorts(sim.resource_out, resources.resource_in)
54         self.connectPorts(resources.resource_out, sim.resource_in)
55         self.connectPorts(check.resource_out, resources.resource_in)
56         self.connectPorts(resources.resource_out, check.resource_in)
57         self.connectPorts(evaluation.resource_out, resources.resource_in)
58         self.connectPorts(resources.resource_out, evaluation.resource_in)

```

---

**Listing 21** Coupled DEVS model of the running example - Part 2.

```
1 def simulate(nresources, verbose, seed=1):
2     random.seed(seed)
3     model = Example(nresources)
4     sim = Simulator(model)
5
6     sim.setClassicDEVS()
7     sim.setTerminationCondition(lambda t, m: m.finish.state is not None)
8     if verbose is not None:
9         sim.setVerbose(verbose)
10    sim.simulate()
11
12    return model.finish.state
13
14 for i in range(1, 16):
15     res = []
16     for v in range(2,100):
17         res.append(simulate(i, None, v))
18     print(str(i) + " " + ".join([str(i) for i in res]))
```

## **Appendix D**

# **Artifacts of the AGV case study**

This appendix documents the artifacts of the AGV case.

**Listing 22** The original design process encoded in Matlab. – Part 1

```

1 function ManualSolution( )
2 clc; close all;
3 %MANUALSOLUTION Show the results of a manual iteration process
4
5 addpath('ComponentSelection');
6 addpath('RSM_Toolbox');
7 addpath('MechanicalDesign');
8
9 MotorDB = MotorDBSetup();
10 BatteryDB = BatteryDBSetup();
11 MotorSelectionF = @(x) MotorSelection(x,MotorDB);
12 BatterySelectionF = @(x) BatterySelection(x,BatteryDB);
13
14 PdesV = 10;
15 CdesV = 1;
16
17 visualize1 = 1;
18 visualize2 = 1;
19 visualize3 = 1;
20 visualize4 = 1;
21 visualize5 = 1;
22 visualize6 = 1;
23 visualize7 = 1;
24 motorAnalytical = 1;
25 batteryAnalytical = 1;
26
27 %% -----
28 % SETUP PROCESS DESCRIPTIONS
29 % -----
30 %% 1. Design steps
31 designSteps = containers.Map();
32     % Motor selection design step
33 motorSelection.parameters = {'Pdes';'Peff';'m';'r';'l'};
34 motorSelection.inputs = {'Pdes'};
35 motorSelection.outputs = {'m';'l';'r';'Peff'};
36 motorSelection.analytical = motorAnalytical;
37 motorSelection.functionCall = '[Peff,m,l,r] = MotorSelectionF(Pdes)';
38 motorSelection.data = containers.Map();
39 designSteps('motorSelection') = motorSelection;
40     % Battery selection design step
41 batterySelection.parameters = {'Cdes';'Ceoff';'m';'l';'w';'h'};
42 batterySelection.inputs = {'Cdes'};
43 batterySelection.outputs = {'m';'l';'w';'h';'Ceoff'};
44 batterySelection.analytical = batteryAnalytical;
45 batterySelection.functionCall = '[Ceoff,m,l,w,h] = BatterySelectionF(Cdes)';
46 batterySelection.data = containers.Map();
47 designSteps('batterySelection') = batterySelection;
48     % Mechanical design design step
49 mechDesign.parameters = {'mBat';'lBat';'wBat';'hBat';'mMot';'lMot';'rMot';...
50     'mTot';'IxxTot';'IyyTot';'IzzTot'};
51 mechDesign.inputs = {'mBat';'lBat';'wBat';'hBat';'mMot';'lMot';'rMot'};
52 mechDesign.outputs = {'mTot';'IxxTot';'IyyTot';'IzzTot'};
53 mechDesign.analytical = 1;
54 mechDesign.functionCall = '[mTot, IxxTot, IyyTot, IzzTot] = MechanicalDesign(mBat,
    ↪ lBat,wBat,hBat,mMot,lMot,rMot)';
55 mechDesign.data = containers.Map();
56 designSteps('mechDesign') = mechDesign;
57     % Controller optimization design step
58 contrOpt.parameters = {'mTot';'IxxTot';'IyyTot';'IzzTot';'PmaxMot';'TavgMot1';
    ↪ 'TavgMot2';'Ku';'Tu'};
59 contrOpt.inputs = {'mTot';'IxxTot';'IyyTot';'IzzTot'};
60 contrOpt.outputs = {'PmaxMot';'TavgMot1';'TavgMot2'};
61 contrOpt.analytical = 0;
62 contrOpt.data = containers.Map();
63 designSteps('contrOpt') = contrOpt;
64     % Electrical Simulation design step
65 elSim.parameters = {'TavgMot1';'TavgMot2';'CBat'};
66 elSim.inputs = {'TavgMot1';'TavgMot2'};
67 elSim.outputs = {'CBat'};
68 elSim.analytical = 0;
69 elSim.data = containers.Map();
70 designSteps('elSim') = elSim;

```

**Listing 23** The original design process encoded in Matlab. – Part 2

---

```

1 %% 2. Connections map
2 sources = containers.Map();
3 sources('mechDesignPoint(''mBat'')') = 'batterySelectionPoint(''m'')'; % [kg]
4 sources('mechDesignPoint(''lBat'')') = 'batterySelectionPoint(''l'')'; % [m]
5 sources('mechDesignPoint(''wBat'')') = 'batterySelectionPoint(''w'')'; % [m]
6 sources('mechDesignPoint(''hBat'')') = 'batterySelectionPoint(''h'')'; % [m]
7 sources('mechDesignPoint(''mMot'')') = 'motorSelectionPoint(''m'')'; % [kg]
8 sources('mechDesignPoint(''lMot'')') = 'motorSelectionPoint(''l'')'; % [m]
9 sources('mechDesignPoint(''rMot'')') = 'motorSelectionPoint(''r'')'; % [m]
10 sources('contrOptPoint(''mTot'')') = 'mechDesignPoint(''mTot'')'; % [kg]
11 sources('contrOptPoint(''IxxTot'')') = 'mechDesignPoint(''IxxTot'')'; % [kg*m^2]
12 sources('contrOptPoint(''IyyTot'')') = 'mechDesignPoint(''IyyTot'')'; % [kg*m^2]
13 sources('contrOptPoint(''IzzTot'')') = 'mechDesignPoint(''IzzTot'')'; % [kg*m^2]
14 sources('elSimPoint(''TavgMot1'')') = 'contrOptPoint(''TavgMot1'')'; % [Nm]
15 sources('elSimPoint(''TavgMot2'')') = 'contrOptPoint(''TavgMot2'')'; % [Nm]
16 assignInputs = @(x) assignInputsF(x,designSteps(x).inputs,sources);
17
18 %% Manual performance of iterations 1 and 2
19 motorSelectionPoint = containers.Map();
20 batterySelectionPoint = containers.Map();
21 mechDesignPoint = containers.Map();
22 contrOptPoint = containers.Map();
23 elSimPoint = containers.Map();

```

---

**Listing 24** The original design process encoded in Matlab. – Part 3

```

1    % ITERATION 1
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
   ↪ ];
4 add(motorSelection.data,motorSelectionPoint);
5
6 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
7 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
   ↪ batterySelectionPoint)];
8 add(batterySelection.data,batterySelectionPoint);
9
10 assignInputs('mechDesign');
11 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
12 add(mechDesign.data,mechDesignPoint);
13
14 assignInputs('contrOpt');
15 contrOptPoint('PmaxMot') = 94.86; % Manually executed [kW]
16 contrOptPoint('TavgMot1') = 6.1266; % Manually executed [Nm]
17 contrOptPoint('TavgMot2') = 3.2249; % Manually executed [Nm]
18 contrOptPoint('Ku') = 0.6952; % Manually executed [1]
19 contrOptPoint('Tu') = 0.1274; % Manually executed [1]
20 add(contrOpt.data,contrOptPoint);
21
22 assignInputs('elSim');
23 elSimPoint('CBat') = 4.56; % [Ah]
24 add(elSim.data,elSimPoint);
25
26 % Generate RSMs
27 motorSelection.RSM = generateRSM(motorSelection);
28 batterySelection.RSM = generateRSM(batterySelection);
29 mechDesign.RSM = generateRSM(mechDesign);
30 contrOpt.RSM = generateRSM(contrOpt);
31 elSim.RSM = generateRSM(elSim);
32
33 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
   ↪ BatteryDB);
34
35 %% Visualize
36 if visualize1
37     visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
   ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
38     fig = gcf;
39     ResizeAndSave( fig,'Column1_Fig1' );
40     saveas(fig,'Column2_Fig1.png');
41     savefig(fig,'Column2_Fig1.fig');
42     saveas(fig,'Column3_Fig1.png');
43     savefig(fig,'Column3_Fig1.fig');
44 end

```

---

**Listing 25** The original design process encoded in Matlab. – Part 4
 

---

```

1 % ITERATION 2
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
4   ↪ ];
4 add(motorSelection.data,motorSelectionPoint);
5
6 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
7 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
8   ↪ batterySelectionPoint)];
8 add(batterySelection.data,batterySelectionPoint);
9
10 assignInputs('mechDesign');
11 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
12 add(mechDesign.data,mechDesignPoint);
13
14 assignInputs('contrOpt');
15 contrOptPoint('PmaxMot') = 110.3; % Manually executed [kW]
16 contrOptPoint('TavgMot1') = 10.3584; % Manually executed [Nm]
17 contrOptPoint('TavgMot2') = 8.7828; % Manually executed [Nm]
18 contrOptPoint('Ku') = 0.6952; % Manually executed [1]
19 contrOptPoint('Tu') = 0.1624; % Manually executed [1]
20 add(contrOpt.data,contrOptPoint);
21
22 assignInputs('elSim');
23 elSimPoint('CBat') = 5.73108; % [Ah]
24 add(elSim.data,elSimPoint);
25
26 % Generate RSMs
27 motorSelection.RSM = generateRSM(motorSelection);
28 batterySelection.RSM = generateRSM(batterySelection);
29 mechDesign.RSM = generateRSM(mechDesign);
30 contrOpt.RSM = generateRSM(contrOpt);
31 elSim.RSM = generateRSM(elSim);
32
33 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
34   ↪ BatteryDB);
35
36 %% Visualize
37 if visualize2
38   visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
39     ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
40   ResizeAndSave( gcf,'Column1_Fig2' );
41 end

```

---

**Listing 26** The original design process encoded in Matlab. – Part 5

```

1 % ITERATION 3
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
4   ↪ ];
4 add(motorSelection.data,motorSelectionPoint);
5
6 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
7 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
8   ↪ batterySelectionPoint)];
8 add(batterySelection.data,batterySelectionPoint);
9
10 assignInputs('mechDesign');
11 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
12 add(mechDesign.data,mechDesignPoint);
13
14 assignInputs('contrOpt');
15 contrOptPoint('PmaxMot') = 92.0; % Manually executed [kW]
16 contrOptPoint('TavgMot1') = 10.2638; % Manually executed [Nm]
17 contrOptPoint('TavgMot2') = 8.8413; % Manually executed [Nm]
18 contrOptPoint('Ku') = 0.6190; % Manually executed [1]
19 contrOptPoint('Tu') = 0.1468; % Manually executed [1]
20 add(contrOpt.data,contrOptPoint);
21
22 assignInputs('elSim');
23 elSimPoint('CBat') = 13.7354; % [Ah]
24 add(elSim.data,elSimPoint);
25
26 % Generate RSMs
27 motorSelection.RSM = generateRSM(motorSelection);
28 batterySelection.RSM = generateRSM(batterySelection);
29 mechDesign.RSM = generateRSM(mechDesign);
30 contrOpt.RSM = generateRSM(contrOpt);
31 elSim.RSM = generateRSM(elSim);
32
33 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
34   ↪ BatteryDB);
35
36 %% Visualize
37 if visualize3
38   visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
39     ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
40   ResizeAndSave( gcf,'Column1_Fig3' );
41 end

```

---

**Listing 27** The original design process encoded in Matlab. – Part 6
 

---

```

1 %% ITERATION 4
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
   ↪ ];
4 add(motorSelection.data,motorSelectionPoint);
5
6 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
7 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
   ↪ batterySelectionPoint)];
8 add(batterySelection.data,batterySelectionPoint);
9
10 assignInputs('mechDesign');
11 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
12 add(mechDesign.data,mechDesignPoint);
13
14 assignInputs('contrOpt');
15 contrOptPoint('PmaxMot') = 98.4; % Manually executed [kW]
16 contrOptPoint('TavgMot1') = 8.9791; % Manually executed [Nm]
17 contrOptPoint('TavgMot2') = 7.7561; % Manually executed [Nm]
18 contrOptPoint('Ku') = 0.5109; % Manually executed [1]
19 contrOptPoint('Tu') = 0.2154; % Manually executed [1]
20 add(contrOpt.data,contrOptPoint);
21
22 assignInputs('elSim');
23 elSimPoint('CBat') = 3.33627; % [Ah]
24 add(elSim.data,elSimPoint);
25
26 % Generate RSMs
27 motorSelection.RSM = generateRSM(motorSelection);
28 batterySelection.RSM = generateRSM(batterySelection);
29 mechDesign.RSM = generateRSM(mechDesign);
30 contrOpt.RSM = generateRSM(contrOpt);
31 elSim.RSM = generateRSM(elSim);
32
33 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
   ↪ BatteryDB);
34
35 %% Visualize
36 if visualize4
37   visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
   ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
38   ResizeAndSave( gcf,'Column1_Fig4' );
39 end

```

---

---

**Listing 28** The original design process encoded in Matlab. – Part 7

---

```

1 %% ITERATION 5
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
4 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
5 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
6 assignInputs('mechDesign');
7 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
8 assignInputs('contrOpt');
9 contrOptPoint('PmaxMot') = 110.3; % Manually executed [kW]
10 contrOptPoint('TavgMot1') = 10.3584; % Manually executed [Nm]
11 contrOptPoint('TavgMot2') = 8.7828; % Manually executed [Nm]
12 contrOptPoint('Ku') = 0.6952; % Manually executed [1]
13 contrOptPoint('Tu') = 0.1624; % Manually executed [1]
14 assignInputs('elSim');
15 elSimPoint('CBat') = 5.73108; % [Ah]
16 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);
17 if visualize5
18     visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
19     ResizeAndSave( gcf,'Column1_Fig5' );
20 end

```

---



---

**Listing 29** The original design process encoded in Matlab. – Part 8

---

```

1 %% ITERATION 6
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
4 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
5 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
6 assignInputs('mechDesign');
7 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
8 assignInputs('contrOpt');
9 contrOptPoint('PmaxMot') = 92.0; % Manually executed [kW]
10 contrOptPoint('TavgMot1') = 10.2638; % Manually executed [Nm]
11 contrOptPoint('TavgMot2') = 8.8413; % Manually executed [Nm]
12 contrOptPoint('Ku') = 0.6190; % Manually executed [1]
13 contrOptPoint('Tu') = 0.1468; % Manually executed [1]
14 assignInputs('elSim');
15 elSimPoint('CBat') = 13.7354; % [Ah]
16 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);
17 if visualize6
18     visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
19     ResizeAndSave( gcf,'Column1_Fig6' );
20 end

```

---

---

**Listing 30** The original design process encoded in Matlab. – Part 9
 

---

```

1 %% ITERATION 7
2 motorSelectionPoint('Pdes') = PdesV(end); % [kW]
3 motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
  ↪ ];
4 batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
5 batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
  ↪ batterySelectionPoint)];
6 assignInputs('mechDesign');
7 mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
8 assignInputs('contrOpt');
9 contrOptPoint('PmaxMot') = 98.4; % Manually executed [kW]
10 contrOptPoint('TavgMot1') = 8.9791; % Manually executed [Nm]
11 contrOptPoint('TavgMot2') = 7.7561; % Manually executed [Nm]
12 contrOptPoint('Ku') = 0.5109; % Manually executed [1]
13 contrOptPoint('Tu') = 0.2154; % Manually executed [1]
14 assignInputs('elSim');
15 elSimPoint('CBat') = 3.33627; % [Ah]
16 [PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
  ↪ BatteryDB);
17 if visualize7
18   visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
  ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
19   ResizeAndSave( gcf,'Columnl_Fig7' );
20 end

```

---

**Listing 31** The original design process encoded in Matlab. – Part 10

```

1 disp('Not done')
2
3 % Assign all inputs
4 function assignInputsF(designstep,properties,sources)
5     for i = 1:length(properties)
6         curProp = properties{i};
7         fullName = [designstep 'Point(' curProp ')'];
8         source = sources(fullName);
9         evalin('caller',[fullName ' = ' source ';']);
10    end
11 end
12 % Add a datapoint to the data set
13 % The data set is a map, with parameter names as keys
14 % The point is a struct, with parameter names as fields
15 function add(data,point)
16     pointParameters = keys(point);
17     nParameters = length(pointParameters);
18     if data.size(1)==0
19         % If the data set is empty, keys are created based in point fields
20         for i = 1:nParameters
21             curKey = pointParameters{i};
22             newData = point(curKey);
23             data(curKey) = newData(:);
24         end
25     else
26         dataParameters = keys(data);
27         nDataPoints = length(data(dataParameters{1}));
28         % If the data is not empty, test that its parameters are the
29         % same as those of the point
30         if ~isempty(setdiff(dataParameters,pointParameters))
31             str = 'Data set uses different parameters as the provided point\n';
32             str = [str 'Data parameters: ' dataParameters '\n'];
33             str = [str 'Point parameters: ' pointParameters '\n'];
34             error(str);
35         end
36         % Check if the distance to the previous points is not too large
37         dataM = zeros(nDataPoints,nParameters);
38         dataV = zeros(1,nParameters);
39         for ii = 1:nParameters
40             dataM(:,ii) = data(dataParameters{ii});
41             dataV(:,ii) = point(dataParameters{ii});
42         end
43         dist = min(sqrt(sum(abs(dataM-ones(nDataPoints,1)*dataV).^2,2)));
44         if dist < 1E-6
45             warning('New data point is too close to an existing point. Ignoring new
46             ↪ point')
47             return;
48         end
49         % Add the data from the point
50         for i = 1:nParameters
51             curKey = pointParameters{i};
52             newData = point(curKey);
53             data(curKey) = [data(curKey); newData(:)];
54         end
55     end

```

---

**Listing 32** The original design process encoded in Matlab. – Part 11
 

---

```

1 % Generate a RSM for the given model with data
2 function rsm = generateRSM(model)
3     %% Extract data
4     nInputs      = length(model.inputs);
5     nOutputs     = length(model.outputs);
6     inputs       = model.inputs;
7     outputs      = model.outputs;
8     nDataPoints  = length(model.data(inputs{1}));
9     dataM = zeros(nDataPoints,nInputs);
10    for ii = 1:nInputs
11        dataM(:,ii) = model.data(inputs{ii});
12    end
13    %% Determine fitting function
14    if nDataPoints >= nInputs+1
15        fittingF = 'regpoly';
16    else
17        fittingF = 'regpoly0';
18    end
19    %% Setup the output struct
20    %% Add which model information was used that may change over time
21    rsm.inputs      = inputs; % Inputs - Outputs may change
22    rsm.outputs     = outputs;
23    rsm.nDataPoints = nDataPoints; % Data may be added
24    %% Fit parameters
25    rsm.dmodel = containers.Map(); % Store a krigin model struct per output
26    for ii = 1:nOutputs
27        %% RSM toolbox: fit Krigin RSM
28        rsm.dmodel(outputs{ii}) = dacefit(dataM,model.data(outputs{ii}), fittingF
29            ↪ , 'correp', ones(1,nInputs), 1e-1*ones(1,nInputs), 20*ones(1,
30            ↪ nInputs));
31    end
32    %% Evaluate a model at a given point
33    %% In case of an analytical design step, it calls the provided function call
34    %% In case of a non-analytical design step, evaluates its RSM
35    function result = evaluateModel(model,point)
36        if model.analytical
37            result = call(model,point);
38        else
39            result = evaluateRSM(model,point);
40        end

```

---

**Listing 33** The original design process encoded in Matlab. – Part 12

```

1  % Evaluate a model using its given function call
2  function result = call(model,point)
3      result = containers.Map();
4      % Extract data
5      nInputs = size(model.inputs,1);
6      nOutputs= size(model.outputs,1);
7      inputs = model.inputs;
8      outputs = model.outputs;
9      functionCall = strrep(model.functionCall,' ','');
10     % Split the function call into inputs, function name and outputs
11     parts      = strsplit(functionCall,'=');
12     outputSs   = strsplit(parts{1}(2:end-1),',');
13     parts      = strsplit(parts{2},')';
14     functionS = parts{1};
15     inputSs = strsplit(parts{2}(1:end-1),',');
16     % Rebuild the function call
17     functionCallParts = cell(0,1);
18     functionCallParts{end+1} = '[';
19     for ii = 1:length(outputSs)
20         functionCallParts{end+1} = outputSs{ii};
21         if ii ~= length(outputSs)
22             functionCallParts{end+1} = ',';
23         end
24     end
25     functionCallParts{end+1} = '[' = ' ' functionS ' (';
26     for ii = 1:length(inputSs)
27         functionCallParts{end+1} = inputSs{ii};
28         if ii ~= length(inputSs)
29             functionCallParts{end+1} = ',';
30         end
31     end
32     functionCallParts{end+1} = ')';'
33     % Rewrite all inputs and outputs to use the provided maps
34     for ii = 1:nInputs
35         for jj = 1:length(functionCallParts)
36             if strcmp(inputs{ii},functionCallParts{jj})
37                 functionCallParts{jj} = ['point(' inputs{ii} ')'];
38                 break;
39             end
40         end
41     end
42     for ii = 1:nOutputs
43         for jj = 1:length(functionCallParts)
44             if strcmp(outputs{ii},functionCallParts{jj})
45                 functionCallParts{jj} = ['result(' outputs{ii} ')'];
46                 break;
47             end
48         end
49     end
50     functionCall = strjoin(functionCallParts,'');
51     % Evaluate the expression
52     eval([functionCall ';' ]);
53 
```

**Listing 34** The original design process encoded in Matlab. – Part 13

---

```

1 % Generate feasibility plot for the given RSMs of the design process
2 function visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,
3     ↪ batterySelection,Plast,Clast)
4     % POWERS
5     pList = [MotorDB.P];
6     pStep = mean(pList(2:end)-pList(1:end-1));
7     pMin = min(pList)-3*pStep;
8     pMax = max(pList);
9     pV = pMin:pStep/2:pMax;
10    % CAPACITIES
11    cList = [BatteryDB.C];
12    cStep = mean(cList(2:end)-cList(1:end-1));
13    cMin = min(cList)-3*cStep;
14    cMax = max(cList);
15    cV = cMin:cStep/2:cMax; %linspace(cMin,cMax,100);
16    feas = zeros(length(pV),length(cV));
17    % Loop over all points
18    for i=1:length(pV)
19        for j = 1:length(cV)
20            feas(i,j) = Plast<=pV(i) && Clast<=cV(j);
21        end
22    end
23    % Plot
24    plotNextStep(PdesV(end-1:end),CdesV(end-1:end),MotorDB,BatteryDB,
25        ↪ motorSelection,batterySelection,cV,pV,feas);
end

```

---

**Listing 35** The original design process encoded in Matlab. – Part 14

```

1 % Generate feasibility plot for the given RSMs of the design process
2 function [curOptP,curOptC] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
3                                     ↪ BatteryDB)
4 % First motor & battery that can provide this
5 curOptP = MotorDB(find([MotorDB.P]>=contrOptPoint('PmaxMot'),1,'first')).P;
6 curOptC = BatteryDB(find([BatteryDB.C]>=elSimPoint('CBat'),1,'first')).C;
7 end
8 % Generate feasibility plot for the given RSMs of the design process
9 function [c,ceq] = evaluateConstraints(Pdes,Cdes,motorSelection,...
10                                batterySelection,mechDesign,contrOpt,elSim)
11
12 % Initialize all input points
13 motorSelectionInput = containers.Map();
14 batterySelectionInput = containers.Map();
15 mechDesignInput = containers.Map();
16 contrOptInput = containers.Map();
17 elSimInput = containers.Map();
18
19 % Initial motor and battery dimensioning
20 motorSelectionInput('Pdes') = Pdes; % [kW]
21 batterySelectionInput('Cdes') = Cdes; % [Ah]
22
23 % Evaluate motor and battery selection models
24 motorSelectionOutput = evaluateModel(motorSelection, motorSelectionInput);
25 batterySelectionOutput = evaluateModel(batterySelection,
26                                     ↪ batterySelectionInput);
27 Peff = motorSelectionOutput('Peff');
28 Ceff = batterySelectionOutput('Ceff');
29
30 % Insert into mechanical design model input
31 mechDesignInput('mBat') = batterySelectionOutput('m');
32 mechDesignInput('lBat') = batterySelectionOutput('l');
33 mechDesignInput('wBat') = batterySelectionOutput('w');
34 mechDesignInput('hBat') = batterySelectionOutput('h');
35 mechDesignInput('mMot') = motorSelectionOutput('m');
36 mechDesignInput('lMot') = motorSelectionOutput('l');
37 mechDesignInput('rMot') = motorSelectionOutput('r');
38
39 % Evaluate mechanical design model
40 mechDesignOutput = evaluateModel(mechDesign, mechDesignInput);
41
42 % Insert into controller optimization input
43 contrOptInput('mTot') = mechDesignOutput('mTot');
44 contrOptInput('IxxTot') = mechDesignOutput('IxxTot');
45 contrOptInput('IyyTot') = mechDesignOutput('IyyTot');
46 contrOptInput('IzzTot') = mechDesignOutput('IzzTot');
47
48 % Evaluate controller optimization model
49 contrOptOutput = evaluateModel(contrOpt, contrOptInput);
50
51 % Insert into controller optimization input
52 elSimInput('TavgMot1') = contrOptOutput('TavgMot1');
53 elSimInput('TavgMot2') = contrOptOutput('TavgMot2');
54
55 % Evaluate controller optimization model
56 elSimOutput = evaluateModel(elSim, elSimInput);
57
58 % Extract required values
59 Preq = contrOptOutput('PmaxMot');
60 Creq = elSimOutput('CBat');
61
62 % Feasibility
63 ceq = [];
64 c = [Preq-Peff; Creq-Ceff];
65
66 end

```

---

**Listing 36** The original design process encoded in Matlab. – Part 15
 

---

```

1  % Visualize a single input-output relationship
2  % Inputs:
3  % - RSM model
4  % - Point containing the value of all other inputs
5  % - Name of the input to vary
6  % - Name of the output to get
7  function visualizeInputOutputRelation(model,point,input,output,fig,opt)
8    % Create range
9    inputData = model.data(input);
10   inputMin = min(inputData)*0.9;
11   inputMax = max(inputData)*1.1;
12   inputVals = linspace(inputMin,inputMax,100);
13   outputVals = zeros(size(inputVals));
14   for ii = 1:length(inputVals)
15     inputVal = inputVals(ii);
16     point(input) = inputVal;
17     result = evaluateModel(model,point);
18     outputVals(ii) = result(output);
19   end
20   figure(fig);
21   hold on;
22   plot(inputVals,outputVals,opt)
23   xlabel(input)
24   ylabel(output)
25 end
26 end

```

---

**Listing 37** The motor database setup activity of the case study.
 

---

```

1 function MotorSelection = MotorSelection()
2 MotorSelection = [];
3 %% Settings
4 powerToWeight = 2; % [kW/kg] --> based on UQM PP 100
5 density = 50/(0.233*(0.286/2)^2*pi); % [kg/m3] --> based on UQM PP 100
6
7 %% Interpolate to create a "limited" database
8 Pvect = 10:10:200; % [kW]
9 for i = 1:length(Pvect)
10   Pcur = Pvect(i);
11   m = Pcur/powerToWeight;
12   r = 0.286/2;
13   l = m/(density*pi*r^2);
14   MotorSelection = addOne(MotorSelection,Pcur,m,l,r);
15 end
16
17 %% Output
18 MotorSelection = orderByField(MotorSelection,'P');
19
20 % Add a single battery
21 function cur = addOne(cur,P,m,l,r)
22   new.P = P;
23   new.m = m;
24   new.l = l;
25   new.r = r;
26   cur = [cur; new];
27 end
28 % Order according to the given field
29 function res = orderByField(in,fieldName)
30   fieldContent = [in.(fieldName)];
31   [~,iOrdering] = sort(fieldContent);
32   res = in(iOrdering);
33 end
34
35 end

```

---

---

**Listing 38** The battery database setup activity of the case study.

---

```

1 function BatterySelection = BatterySelection()
2 BatterySelection = []
3 %% Load data
4 dataArray = csvread('KokamLargeBatteries.csv',1,0);
5 C.data = dataArray(:, 1); % [Ah]
6 w.data = dataArray(:, 2)./1000; % [m]
7 l.data = dataArray(:, 3)./1000; % [m]
8 t.data = dataArray(:, 4)./1000; % [m]
9 m.data = dataArray(:, 5); % [kg]
10
11 %% Create least-squares model
12 w.coef = [ones(size(C.data)) C.data]\w.data;
13 l.coef = [ones(size(C.data)) C.data]\l.data;
14 t.coef = [ones(size(C.data)) C.data]\t.data;
15 m.coef = [ones(size(C.data)) C.data]\m.data;
16
17 %% Interpolate to create a "limited" database
18 Cvect = 2:10:102;
19 for Ccur = Cvect
20     BatterySelection = addOne(BatterySelection,Ccur, ...
21         [1 Ccur]*m.coef, ...
22         [1 Ccur]*w.coef, ...
23         [1 Ccur]*l.coef, ...
24         [1 Ccur]*t.coef);
25 end
26
27 %% Output
28 BatterySelection = orderByField(BatterySelection,'C');
29
30 % Add a single battery
31 function cur = addOne(cur,C,m,l,w,h)
32     new.C = C;
33     new.m = m;
34     new.l = l;
35     new.w = w;
36     new.h = h;
37     cur = [cur; new];
38 end
39 % Order according to the given field
40 function res = orderByField(in,fieldName)
41     fieldContent = [in.(fieldName)];
42     [~,iOrdering] = sort(fieldContent);
43     res = in(iOrdering);
44 end
45
46 end
```

---

---

**Listing 39** The original source of the battery DB.
 

---

```

1 Capacity,w,l,t,m
2 13,220,132,6.3,0.332
3 12,220,132,6.5,0.34
4 16,220,132,7.8,0.406
5 25,226,227,6.3,0.6
6 31,226,227,7.5,0.72
7 40,226,227,9.3,0.9
8 46,226,227,12.5,1.265
9 53,226,227,12,1.16
10 30,198,220,9.9,0.84
11 52,268,265,9.3,1.26
12 55,268,265,10.3,1.45
13 63,268,265,11,1.52
14 75,268,265,12,1.63
15 87,268,265,13,1.78
16 65,268,265,13.5,1.85
17 75,268,265,13.3,2
18 70,462,327,5.4,1.63
19 80,462,327,6.3,1.92
20 100,462,327,7,2.07
21 150,462,327,10.5,3.21
22 200,462,327,13.7,4.18
23 240,462,327,15.8,4.78

```

---

**Listing 40** The battery selection activity of the case study.
 

---

```

1 function [ Ceff,m,l,w,h ] = BatterySelection( Cdes,BatteryDB )
2 %BATTERYSELECTION
3
4 iBat = find([BatteryDB.C]>=Cdes,1,'first');
5 Ceff = BatteryDB(iBat).C; % [Ah]
6 m = BatteryDB(iBat).m; % [kg]
7 l = BatteryDB(iBat).l; % [m]
8 w = BatteryDB(iBat).w; % [m]
9 h = BatteryDB(iBat).h; % [m]
10
11 end

```

---

**Listing 41** The motor selection activity of the case study.
 

---

```

1 function [ Peff,m,l,r ] = MotorSelection( Pdes, MotorDB )
2 %MOTORSELECTION
3
4 iMot = find([MotorDB.P]>=Pdes,1,'first');
5 Peff = MotorDB(iMot).P; % [kW]
6 m = MotorDB(iMot).m; % [kg]
7 l = MotorDB(iMot).l; % [m]
8 r = MotorDB(iMot).r; % [m]
9
10 end

```

---

**Listing 42** The mechanical design and simulation activity of the case study.

```

1 function [mTot, IxxTot, IyyTot, IzzTot] = MechanicalDesign(mBat,lBat,wBat,hBat, ...
2     mMot,lMot,rMot)
3
4 % Assumptions:
5 % Battery is placed at the center of the robot
6 % length = y-axis, width = x-axis, height = z-axis
7 Bat.x = 0;
8 Bat.y = 0;
9 Bat.z = hBat/2;
10 Bat.m = mBat;
11 elements = {Bat};
12 % Motors are placed at opposite ends of the battery
13 % parallel to the y-axis, perpendicular to the x and z-axes
14 Mot1.x = 0;
15 Mot1.y = wBat/2+lMot/2;
16 Mot1.z = rMot;
17 Mot1.m = mMot;
18 Mot2.x = 0;
19 Mot2.y = -(wBat/2+lMot/2);
20 Mot2.z = rMot;
21 Mot2.m = mMot;
22 elements{end+1} = Mot1;
23 elements{end+1} = Mot2;
24 % Magnets are placed next to the batteries in the x-direction
25 Mag1.x = lBat/2+0.087/2;
26 Mag1.y = 0;
27 Mag1.z = 0;
28 Mag1.m = 2.989;
29 Mag2.x = -(lBat/2+0.087/2);
30 Mag2.y = 0;
31 Mag2.z = 0;
32 Mag2.m = 2.989;
33 elements{end+1} = Mag1;
34 elements{end+1} = Mag2;
35 % Base is a circle,
36 % with a plate thickness determined by the battery and motor mass
37 Base.x = 0;
38 Base.y = 0;
39 Base.z = 0;
40 Base.rho = 7800;
41 Base.r = wBat/2+lMot;
42
43 mOnBase = 0;
44 for curEl = elements
45     mOnBase = mOnBase + curEl{1}.m;
46 end
47 Base.t = 3*mOnBase/1E5; % Linear approximation of the thickness
48
49 Base.m = Base.rho * pi * Base.r^2 * Base.t;
50 Base.Ixx = 1/12*Base.m*(3*Base.r^2+Base.t^2);
51 Base.Iyy = 1/12*Base.m*(3*Base.r^2+Base.t^2);
52 Base.Izz = 1/2*Base.m*Base.r^2;
53
54 elements{end+1} = Base;
55 % Sum all elements
56 mTot = 0; IxxTot = Base.Ixx; IyyTot = Base.Iyy; IzzTot = Base.Izz;
57 for curEl = elements
58     mTot = mTot + curEl{1}.m;
59     IxxTot = IxxTot + curEl{1}.m*(curEl{1}.y^2+curEl{1}.z^2);
60     IyyTot = IyyTot + curEl{1}.m*(curEl{1}.x^2+curEl{1}.z^2);
61     IzzTot = IzzTot + curEl{1}.m*(curEl{1}.x^2+curEl{1}.y^2);
62 end
63
64 end

```

**Listing 43** The exported plant model from Virtual.Lab-Motion to Matlab/Simulink.

```

1 motionfiles =[  

2 'C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1,  

3 C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1,  

4 C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1,  

5 C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1'];  

6 antype = ['cosim']; feedthrough = ['false'];  

7 vl = getenv('VLMOTIONS LV');  

8 vll = strcat(vl,'execute\intel64');  

9 path(path,vll);  

10 if ( strcmp(gcs,'') )  

11     disp('Error Matlab model not loaded');  

12     clear vl vll;  

13     return;  

14 end  

15 subsys = strcat(gcs,'/plantout');  

16 new = 0;  

17 try  

18     add_block('built-in/SubSystem', subsys, 'Position', [200 200 300 300] );  

19     new = 1;  

20 catch  

21     try  

22         cnt = get_param( strcat(subsys,'/thedemux') , 'Outputs' );  

23         for i = 1:str2num(cnt)  

24             delete_line( subsys, strcat('thedemux//',num2str(i)), strcat('themux//',  

25             ↪ num2str(i)) );  

26         end  

27         delete_line(subsys,'in/1', 'thebus/1' );  

28         delete_line(subsys,'thebus/1','motionfun/1' );  

29         delete_line(subsys,'themux/1','out/1');  

30         delete_line(subsys,'motionfun/1','thedemux/1');  

31         delete_block( strcat(subsys,'/motionfun') );  

32         delete_block( strcat(subsys,'/thedemux') );  

33         delete_block( strcat(subsys,'/themux') );  

34         delete_block( strcat(subsys,'/thebus') );  

35     catch  

36         warning 'Unable to modify plantout block will delete';  

37         delete_block( subsys );  

38         add_block('built-in/SubSystem', subsys, 'Position', [200 200 300 300] );  

39         new = 1;  

40     end  

41 end  

42 if ( new )  

43     add_block('built-in/Inport', strcat( subsys, '/in'), 'Position', [15 125 65 175] )  

44     ↪ ;  

45     add_block('built-in/Outport', strcat( subsys, '/out'), 'Position', [735 125 785  

46     ↪ 175] );  

47 end  

48 add_block('built-in/Demux', strcat(subsys,'/thedemux'), 'Outputs', '2', 'Position',  

49     ↪ [365 100 465 200] );  

50 add_block('built-in/BusCreator', strcat(subsys,'/themux'), 'Inputs', '2', 'Position',  

51     ↪ [580 100 590 200] );  

52 add_block('built-in/BusSelector', strcat(subsys,'/thebus'), 'Position', [120 100 130  

53     ↪ 200], 'MuxedOutPut', 'On' );  

54 sfunction_setup = 1;  

55 add_block('built-in/S-Function', strcat(subsys,'/motionfun'), 'Position', [210 100  

56     ↪ 290 200], 'FunctionName', 'vlmotionmex', 'Parameters', 'antype,motionfiles,  

57     ↪ feedthrough' );  

58 clear sfunction_setup;  

59 add_line(subsys,'themux/1','out/1');  

60 add_line(subsys,'in/1','thebus/1');  

61 add_line(subsys,'thebus/1','motionfun/1');  

62 add_line(subsys,'motionfun/1','thedemux/1');  

63 a = add_line( subsys, 'thedemux/1', 'themux/1' );  

64 set_param( a, 'Name', 'a1' );  

65 a = add_line( subsys, 'thedemux/2', 'themux/2' );  

66 set_param( a, 'Name', 'a2' );  

67 set_param( strcat(subsys,'/thebus'), 'OutputSignals', 'T1,T2' );  

68 clear vl vll a i new subsys;  

69 disp('setup for Virtual.Lab Motion done');

```

---

**Listing 44** The querying template for AMESim.

```

1 class AmesimQuery extends AmesimAttributeDefinitionImpl implements IExecutable {
2
3     private val logger = Logger.getLogger(this.class)
4     private val AMESIM_LOCATION = new File("D:\\tools\\LMS\\LMS Imagine.Lab Amesim\\\
5         ↪ v1400")
6     private val QUERY_RESULT_LABEL = "queryresult:"
7     private val ERROR_MSG = "From AMEGetParameterValue: fail to locate parameter or
8         ↪ variable in the active circuit."
9
10    @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Attribute attribute;
11    @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Activity activity;
12
13    new(Attribute attribute, Activity activity) {
14        this.attribute = attribute
15        this.activity = activity
16    }
17
18    override execute() {
19        logger.level = Level::DEBUG
20
21        val file = File.createTempFile("amesimquery", ".py", AMESIM_LOCATION)
22        file.deleteOnExit()
23        val bufferedWriter = new BufferedWriter(new FileWriter(file))
24
25        val scriptText = AmesimQueryScriptTemplate.generateAmesimQueryScript(attribute.
26            ↪ getQueryableName(activity)).toString
27        bufferedWriter.write(scriptText)
28        bufferedWriter.close();
29
30        logger.debug(file.getAbsolutePath())
31        logger.debug(scriptText)
32
33        val command = '"' + AMESIM_LOCATION + '\\python.bat" "' + file.getAbsolutePath() + '"'
34        logger.debug('command: ' + command)
35
36        val processBuilder = new ProcessBuilder(command)
37        processBuilder.directory(AMESIM_LOCATION)
38        val process = processBuilder.start
39
40        val bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream
41            ↪ ))
42        val errorReader = new BufferedReader(new InputStreamReader(process.getErrorStream()))
43
44        evaluate(bufferedReader, errorReader)
45    }
46
47    private def evaluate(BufferedReader bufferedReader, BufferedReader errorReader) {
48        val List<String> blines = Lists::newArrayList(bufferedReader.lines().toArray)
49        if (!blines.empty) {
50            for (line : blines) {
51                logger.debug(line)
52                if (line.trim.startsWith(QUERY_RESULT_LABEL)) {
53                    return Double.parseDouble(line.split(':')[].last.trim)
54                }
55            }
56        }
57    }
58}
```

---

---

**Listing 45** The querying template for Matlab.
 

---

```

1 class MatlabQuery extends MatlabAttributeDefinitionImpl implements IExecutable {
2
3     private static extension val AliasHandler aliasHandler = new AliasHandler()
4
5     @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Attribute attribute;
6     @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Activity activity;
7
8     new(Attribute attribute, Activity activity) {
9         this.attribute = attribute
10        this.activity = activity
11    }
12
13    override execute() {
14        try {
15            val matlabEngine = MatlabConnectionManager::matlabEngine;
16            val queryableName = attribute.getQueryableName(activity)
17            matlabEngine.getVariable(queryableName)
18        } catch (Exception ex) {
19            return null
20        }
21    }
22 }
```

---

**Listing 46** The executor for Python.
 

---

```

1 class PythonExecutor {
2
3     def execute(PythonScript script) {
4         //val p = Runtime.getRuntime().exec("d:\\GitHub\\msdl\\ICM\\examples\\be.
5         ↳ uantwerpen.msdl.icm.cases.demo\\python\\pythonstep.bat amesimtest.py")
6         //this was added due to the standalone python installation packaged with LMS
7         ↳ tools (e.g. AMESIM)
8         val p = Runtime.getRuntime().exec("python " + script.getScriptLocation());
9         val in = new BufferedReader(new InputStreamReader(p.getInputStream()));
10        System.out.println(in.readLine());
11    }
12 }
```

---

**Listing 47** The command executor for Matlab extending the *ParameterizedExecutor* in Listing 48.

```

1 enum ExecutionMode {
2     NORMAL,
3     HEADLESS
4 }
5
6 class MatlabExecutor extends ParameterizedExecutor {
7     @Accessors(NONE) val DEFAULT_MODE = ExecutionMode::HEADLESS
8
9     def execute(MatlabScript script, MatlabProxy matlabProxy, EMap<String, String>
10         ↪ parameters,
11         ExecutionMode executionMode) {
12         switch (executionMode) {
13             case NORMAL: executeWithGui(script, matlabProxy, parameters)
14             case HEADLESS: executeHeadless(script, matlabProxy, parameters)
15         }
16     }
17
18     def execute(MatlabScript script, MatlabProxy matlabProxy, EMap<String, String>
19         ↪ parameters) {
20         execute(script, matlabProxy, parameters, DEFAULT_MODE)
21     }
22
23     private def executeHeadless(MatlabScript script, MatlabProxy matlabProxy, EMap<
24         ↪ String, String> parameters) {
25         val path = Paths.get(script.scriptLocation)
26         val charset = StandardCharsets.UTF_8
27         val rawContent = new String(Files.readAllBytes(path), charset)
28
29         // resolve parameters in the next line of the script
30         val content = rawContent.resolveParameters(parameters)
31
32         // execute command
33         val matlabEngine = MatlabConnectionManager::matlabEngine
34         matlabEngine.eval(content)
35     }
36
37     private def executeWithGui(MatlabScript script, MatlabProxy matlabProxy, EMap<
38         ↪ String, String> parameters) {
39         val file = new File(script.scriptLocation)
40
41         try {
42             val bufferedReader = new BufferedReader(new FileReader(file))
43             var String line = ""
44
45             while ((line = bufferedReader.readLine()) != null) {
46                 // resolve parameters in the next line of the script
47                 line = line.resolveParameters(parameters)
48
49                 // execute command
50                 matlabProxy.eval(line)
51
52                 // update variable store
53                 line.extractAssignments
54             }
55         } catch (Exception e) {
56             e.printStackTrace
57         }
58     }
59 }
```

**Listing 48** The parameterized command executor. Serves as a superclass for platform-specific executors.

```

1 class ParameterizedExecutor {
2
3     val BYNUMBER_NAME_PATTERN = "\\\\$\\{args\\[i\\].name\\}\\\$"
4     val BYNUMBER_VALUE_PATTERN = "\\\\$\\{args\\[i\\].value\\}\\$"
5     val BYNAME_NAME_PATTERN = "\\\\$\\{args\\['n'\\].name\\}\\$"
6     val BYNAME_VALUE_PATTERN = "\\\\$\\{args\\['n'\\].value\\}\\$"
7
8     val ASSIGNMENT_PATTERN = "varname\\\$*=\\\$*[0-9]*"
9
10    def resolveParameters(String command, EMap<String, String> parameters) {
11        var String newCommand = command
12
13        for (parameter : parameters) {
14            val key = parameter.key
15            val value = parameter.value
16            val index = parameters.indexOfKey(key)
17
18            newCommand = newCommand.replaceAll(BYNUMBER_NAME_PATTERN.replace('i', index.
19                ↳ toString), key)
20            newCommand = newCommand.replaceAll(BYNUMBER_VALUE_PATTERN.replace('i', index.
21                ↳ toString), value)
22            newCommand = newCommand.replaceAll(BYNAME_NAME_PATTERN.replaceFirst('n', key),
23                ↳ key)
24            newCommand = newCommand.replaceAll(BYNAME_VALUE_PATTERN.replaceFirst('n', key),
25                ↳ value)
26        }
27
28        newCommand = newCommand.replaceUnused
29
30        newCommand
31    }
32
33    def getReplaceUnused(String command) {
34        var String newCommand = command
35        newCommand = newCommand.replaceAll("\\\\$\\{args\\['[a-zA-Z0-9_]*'\\].value\\}\\$",
36            ↳ 'null')
37        newCommand
38    }
39
40    def extractAssignments(String line) {
41        val variableManager = VariableManager.instance
42
43        for (variable : variableManager.variableStore.variables) {
44            val pattern = ASSIGNMENT_PATTERN.replace('varname', variable.name)
45            val matches = Pattern.compile(pattern).matcher(line)
46            while (matches.find) {
47                val split = matches.group.split("=");
48                val varname = split.head
49                val value = new Double(split.last)
50
51                variableManager.setVariable(varname, value)
52            }
53        }
54    }
55}
```



# Bibliography

- [1] Merriam-Webster Dictionary and Thesaurus. <http://merriam-webster.com>. Accessed: 2015-11-20. 25
- [2] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth. Multi-objective optimization in rule-based design space exploration. In *Proceedings of the 29th ACM/IEEE Int. Conf. on Automated software engineering*, pages 289–300. ACM, 2014. 82
- [3] Accreditation Board for Engineering and Technology Inc. ABET Definition of Design. <http://www.me.unlv.edu/Undergraduate/coursenotes/meg497/ABETdefinition.htm>. Acc: 2017-08-17. 17
- [4] C. Adourian and H. Vangheluwe. Consistency between geometric and dynamic views of a mechanical system. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC ’07, pages 31:1–31:6, San Diego, CA, USA, 2007. Society for Computer Simulation International. 30, 32, 38, 39, 40
- [5] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. 18, 139, 180
- [6] M. A. Almeida da Silva, R. Bendraou, X. Blanc, and M.-P. Gervais. *Early Deviation Detection in Modeling Activities of MDE Processes*, pages 303–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. 104
- [7] A. Alshareef, H. S. Sarjoughian, and B. Zarrin. An approach for activity-based devs model specification. In *Proceedings of the Symposium on Theory of Modeling & Simulation*, TMS-DEVS ’16, pages 25:1–25:8, San Diego, CA, USA, 2016. Society for Computer Simulation International. 46
- [8] Apache Software Foundation. Apache Maven Website. <https://maven.apache.org>. Acc: 2017-08-17. 19
- [9] C. Artigues, S. Demassey, and E. Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2007. 47, 73, 138
- [10] C. Atkinson and D. Draheim. Cloud-aided software engineering: evolving viable software systems through a web of views. In *Software engineering frameworks for the cloud computing paradigm*, pages 255–281. Springer, 2013. 10

- [11] S. Balaji and M. S. Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012. 18
- [12] R. Balzer. Tolerating inconsistency. *[1991 Proceedings] 13th International Conference on Software Engineering*, pages 158–165, 1991. 30, 39, 41, 129, 177, 179
- [13] B. Barragáns-Martínez, J. Pazos-Arias, and A. Fernández-Vilas. On measuring levels of inconsistency in multi-perspective requirements specifications. In *Proceedings of the 1st Conference on the Principles of Software Engineering (PRISE'04)*, pages 21–30, 2004. 42
- [14] B. Barroca, T. Kuhne, and H. Vangheluwe. Integrating language and ontology engineering. In *Proceedings of 8th International Workshop on Multi-paradigm modeling*, pages 77–86, 2014. 180
- [15] S. Bechhofer. OWL: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009. 71
- [16] S. M. Becker and A.-T. Körtgen. Integration tools for consistency management between design documents in development processes. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-driven Engineering*, pages 683–718. Springer-Verlag, Berlin, Heidelberg, 2010. 30, 32, 38, 39
- [17] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen. Contracts for Systems Design : Theory. Technical Report RR-8759, INRIA, 2015. 54, 55, 136, 140
- [18] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. VIATRA 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, pages 101–110. Springer, 2015. 83, 91
- [19] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. 42
- [20] Berx, Kristof and Karhula, Henri and Nicolai, Mike and Maes Davy. Extended dependency modelling for effective and efficient design decision support. (Technical report). 61
- [21] J. Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004. 11
- [22] A. K. Bhattacharjee and R. K. Shyamasundar. Activity diagrams : A formal framework to model business processes and code generation. *Journal of Object Technology*, 8(1):189–220, 2009. 45
- [23] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. Multi-domain modeling of cyber-physical systems using architectural views. *AVICPS 2010*, page 43, 2010. 30, 34, 39

- [24] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. View consistency in architectures for cyber-physical systems. In *Cyber-Physical Systems (ICCPs), 2011 IEEE/ACM International Conference on*, pages 151–160, April 2011. 30, 35, 39
- [25] G. Bianchi, F. Paolucci, P. Van den Braembussche, H. Van Brussel, and F. Jovane. Towards virtual engineering in machine tool design. *CIRP Annals-Manufacturing Technology*, 45(1):381–384, 1996. 10
- [26] G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22–27, 2009. 170, 175
- [27] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 511–520, May 2008. 30, 35, 38, 39, 41
- [28] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988. 18
- [29] N. Bouillot and E. Gressier-Soudan. Consistency models for distributed interactive multimedia applications. *SIGOPS Operating Systems Review*, 38(4):20–32, Oct. 2004. 42
- [30] G. Box, G. Jenkins, and G. Reinsel. *Time Series Analysis: Forecasting and Control*. Wiley Series in Probability and Statistics. Wiley, 2008. 139
- [31] G. E. Box. Robustness in the strategy of scientific model building. In *Robustness in statistics*, pages 201–236. Elsevier, 1979. 11
- [32] D. Broman, E. A. Lee, S. Tripakis, and M. Törngren. Viewpoints, formalisms, languages, and tools for cyber-physical systems. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 49–54. ACM, 2012. 15, 56
- [33] B. Choudhary. *The Elements of Complex Analysis*. New Age International, 1992. ISBN 978-81-224-0399-2. 106
- [34] J. O. Clark. System of systems engineering and family of systems engineering from a standards, v-model, and dual-v model perspective. In *2009 3rd Annual IEEE Systems Conference*, pages 381–387. IEEE, 2009. 18
- [35] CMMI Institute. CMMI. <http://cmmiinstitute.com/capability-maturity-model-integration>. Accessed: 2018-11-20. 9, 91, 106, 178
- [36] G. Cohen, D. Dubois, J. Quadrat, and M. Viot. A Linear-System-Theoretic View of Discrete-Event Processes and its use for Performance Evaluation in Manufacturing. *IEEE Transactions on Automatic Control*, 30(3):210–220, Mar 1985. 47
- [37] M. Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004. 10
- [38] J. Corley, E. Syriani, H. Ergin, and S. Van Mierlo. Cloud-based multi-view modeling environments. In *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, pages 120–139. IGI Global, 2016. 16, 129

- [39] M. Cumberlidge. *Business process management with JBoss jBPM*. Packt Publishing Ltd, 2007. 45
- [40] K. Dahman, F. Charoy, and C. Godart. Towards consistency management for a business-driven development of soa. In *Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International*, pages 267–275, Aug 2011. 30, 33, 39, 40
- [41] I. Dávid, J. Denil, K. Gadeyne, and H. Vangheluwe. Engineering Process Transformation to Manage (In)consistency. In *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMITMDE 2016)*, pages 7–16. <http://ceur-ws.org/Vol-1717/>, 2016. 63, 78, 101, 108, 119, 128
- [42] I. Dávid, J. Denil, and H. Vangheluwe. Towards inconsistency management by process-oriented dependency modeling. In *Proceedings of 9th International Workshop on Multi-Paradigm Modeling*, pages 32–41, 2015. 63, 68, 99, 119, 138
- [43] I. Dávid, B. Meyers, K. Vanherpen, Y. Van Tendeloo, K. Berx, and H. Vangheluwe. Modeling and enactment support for managing inconsistencies in heterogeneous systems engineering processes. In *Proceedings of MODELS 2017 Satellite Event, September 17, 2017, Austin, Texas, USA/Burgueño, Loli [edit.]*, pages 145–154, 2017. 91, 95
- [44] I. Dávid, I. Ráth, and D. Varró. Foundations for Streaming Model Transformations by Complex Event Processing. *Software and Systems Modeling*. 40, 139
- [45] I. Dávid, E. Syriani, C. Verbrugge, D. Buchs, D. Blouin, A. Cicchetti, and K. Vanherpen. Towards inconsistency tolerance by quantification of semantic inconsistencies. *1st International Workshop on Collaborative Modelling in MDE*, 2016. 129
- [46] I. Dávid, Y. Van Tendeloo, and H. Vangheluwe. Translating engineering workflow models to devs for performance evaluation. In *Proceedings of the 2018 Winter Simulation Conference*, WSC 2018, page ??? IEEE, Dec. 2018. 106
- [47] J. de Lara and H. Vangheluwe. AToM3: A Tool for Multi-formalism and Metamodelling. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering: 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, pages 174–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. 138
- [48] K. Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014. 155
- [49] J. Denil, R. Salay, C. Paredis, and H. Vangheluwe. Towards agile model-based systems engineering. In *CEUR workshop proceedings*, pages 424–429, 2017. 9
- [50] Denil, Joachim. Design, verification and deployment of software-intensive systems: a multi-paradigm modelling approach. *University Antwerp*, 2013. 92
- [51] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren. Cyber-Physical System Design Contracts. In *ICCPs '13*, pages 109–118. ACM, 2013. 136

- [52] M. Dumas and A. H. Ter Hofstede. Uml activity diagrams as a workflow specification language. In *International conference on the unified modeling language*, pages 76–90. Springer, 2001. 19
- [53] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998. 139
- [54] I. Dávid, J. Denil, and H. Vangheluwe. Patterns of inconsistency management in mechatronics – a survey. (Technical report, 2015). <http://msdl.cs.mcgill.ca/people/istvan/pub/icm-patterns-techreport>. 21
- [55] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating distributed viewpoints: the anatomy of a consistency check. *Concurrent Engineering*, 2(3):209–222, 1994. 30, 39, 41
- [56] Eclipse Foundation. Eclipse Modeling Framework (EMF) Website. <https://eclipse.org/modeling/emf/>. Acc: 2017-08-17. 91, 162
- [57] Eclipse Foundation. EMFCompare. <https://www.eclipse.org/emf/compare/>. Accessed: 2015-11-20. 38
- [58] Eclipse Foundation. MWE2 Website. <https://help.eclipse.org/luna/topic/org.eclipse.xtext.doc/contents/118-mwe-in-depth.html>. Acc: 2017-08-17. 19
- [59] Eclipse Foundation. Sirius Website. <https://eclipse.org/sirius/>. Acc: 2017-07-07. 65, 98, 163
- [60] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, March 2011. 30, 33, 38, 39, 104
- [61] M. El Hamlaoui, S. Ebersold, B. Coulette, M. Nassar, and A. Anwar. Heterogeneous models matching for consistency management. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–12, May 2014. 30, 34, 38, 39
- [62] G. Engels, R. Heckel, and J. Küster. The consistency workbench: A tool for consistency management in uml-based development. In P. Stevens, J. Whittle, and G. Booch, editors, «UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications, volume 2863 of *Lecture Notes in Computer Science*, pages 356–359. Springer Berlin Heidelberg, 2003. 30, 36, 39
- [63] Engineers' Council for Professional Development. Canons of ethics for engineers. <http://www.worldcat.org/title/canons-of-ethics-for-engineers/oclc/26393909>. Accessed: 2018-11-20. 7
- [64] S. D. Eppinger and T. R. Browning. *Design structure matrix methods and applications*. MIT press, 2012. 164

- [65] R. Eramo, A. Pierantonio, and G. Rosa. Approaching Collaborative Modeling as an Uncertainty Reduction Process. In *COMMItMDE@ MoDELS*, pages 27–34, 2016. 104
- [66] C. Ericson. *Real-time collision detection*. CRC Press, 2004. 132
- [67] J.-M. Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*, pages 262–271. Citeseer, 2004. 12, 13
- [68] S. Feldmann, K. Kernschmidt, and B. Vogel-Heuser. Combining a SysML-based Modeling Approach and Semantic Technologies for Analyzing Change Influences in Manufacturing Plant Models. *Procedia /CIRP*, 17:451 – 456, 2014. 100
- [69] D. Fensel. Ontologies. In *Ontologies*, pages 11–18. Springer, 2001. 179
- [70] A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In M. Ibrahim, J. Küng, and N. Revell, editors, *Database and Expert Systems Applications*, volume 1873 of *Lecture Notes in Computer Science*, pages 1–5. Springer Berlin Heidelberg, 2000. 3, 26, 31, 41, 51, 129
- [71] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, Aug 1994. 1, 17
- [72] J. W. Forrester. *Principles of Systems*. Productivity Press, 1968. 71, 99
- [73] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 152–163, 2003. 45
- [74] M. Fowler. *Domain-specific languages*. Pearson Education, 2010. 13
- [75] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In L. Briand and A. L. Wolf, editors, *Future of Software Engineering*, pages 37–54, Minneapolis, may 2007. IEEE Computer Society. 1
- [76] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 621–630, 2004. 45
- [77] D. M. Gabbay, I. Hodkinson, and M. Reynolds. Temporal logic mathematical foundations and computational aspects. 1994. 139
- [78] J. Gausemeier, W. Schäfer, J. Greenyer, S. Kahl, S. Pook, J. Rieke, et al. Management of cross-domain model consistency during the development of advanced mechatronic systems. *DS 58-6: Proceedings of ICED 09, the 17th International Conference on Engineering Design, Vol. 6, Design Methods and Tools (pt. 2), Palo Alto, CA, USA, 24.-27.08. 2009*, 2009. 1, 30, 32, 38, 39, 40
- [79] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on*

- Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 179–188, New York, NY, USA, 2004. ACM. 30, 36, 39
- [80] H. Giese and S. Hildebrandt. Incremental model synchronization for multiple updates. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRAMoT '08, pages 1–8, New York, NY, USA, 2008. ACM. 30, 33, 38, 39, 40
- [81] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 543–557. Springer Berlin Heidelberg, 2006. 33
- [82] D. Gross, J. Shortle, J. Thompson, and C. Harris. *Fundamentals of Queueing Theory*. Wiley, 2011. 88
- [83] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, Nov 1998. 1, 17
- [84] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007. 27
- [85] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. 19, 45, 120
- [86] A. Hegedüs, A. Horvath, I. Rath, M. Branco, and D. Varro. Quick fix generation for dsmls. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 17–24, Sept 2011. 40, 44
- [87] Á. Hegedüs, Á. Horváth, and D. Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering*, 22(3):399–436, 2015. 82, 86
- [88] P. Hehenberger, A. Egyed, and K. Zeman. Consistency checking of mechatronic design models. In *ASME 2010 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1141–1148. American Society of Mechanical Engineers, 2010. 2, 26, 30, 36, 38, 39
- [89] S. J. Herzig and C. J. Paredis. Bayesian reasoning over models. In *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVA 2014*, page 69, 2014. 30, 37, 39
- [90] S. J. Herzig and C. J. Paredis. A conceptual basis for inconsistency management in model-based systems engineering. *Procedia {CIRP}*, 21:52 – 57, 2014. 24th {CIRP} Design Conference. 25, 26, 130
- [91] S. J. Herzig, A. Qamar, and C. J. Paredis. An approach to identifying inconsistencies in model-based systems engineering. *Procedia Computer Science*, 28:354–362, 2014. 1
- [92] A. Hessellund, K. Czarnecki, and A. Wąsowski. Guided development with multiple domain-specific languages. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2007. 30, 31, 38, 39, 40, 44

- [93] G. Huang, K. Bryden, and D. McCorkle. Interactive design using cfd and virtual engineering. In *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 4364, 2004. 10
- [94] A. Hunter, S. Konieczny, et al. Measuring inconsistency through minimal inconsistent sets. *KR*, 8:358–366, 2008. 42
- [95] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille. Consistency problems in uml-based software development. In *Proceedings of the 2004 International Conference on UML Modeling Languages and Applications, UML'04*, pages 1–12, Berlin, Heidelberg, 2005. Springer-Verlag. 26
- [96] P. Höfner. How to find algebraic semantics for modal and temporal logics. [https://www.informatik.uni-augsburg.de/lehrstuehle/dbis/pmi/alumni/hoefner/talks/09\\_relmics/](https://www.informatik.uni-augsburg.de/lehrstuehle/dbis/pmi/alumni/hoefner/talks/09_relmics/). Invited talk at the 11th International Conference on Relational Methods in Computer Science / 6th International Conference on Applications of Kleene Algebra (RelMiCS11/AKA6). Accessed: 2016-08-30. 139
- [97] International Council on Systems Engineering. Additional Methodologies Identified as Gaps since 2008 INCOSE Survey. <http://www.omgwiki.org/MBSE/doku.php?id=mbse:methodology>. Accessed: 2018-11-20. 7
- [98] International Council on Systems Engineering. Survey of Model-Based Systems Engineering (MBSE) Methodologies. [https://oldsite.incose.org/ProductsPubs/pdf/techdata/MTTC/MBSE\\_Methodology\\_Survey\\_2008-0610\\_RevB-JAE2.pdf](https://oldsite.incose.org/ProductsPubs/pdf/techdata/MTTC/MBSE_Methodology_Survey_2008-0610_RevB-JAE2.pdf). Accessed: 2018-11-20. 7
- [99] International Council on Systems Engineering. Systems Engineering Vision 2020. [http://oldsite.incose.org/ProductsPubs/pdf/SEVision2020\\_20071003\\_v2\\_03.pdf](http://oldsite.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf). Accessed: 2018-11-20. 7
- [100] International Council on Systems Engineering. What is Systems Engineering? <https://www.incose.org/AboutSE/WhatIsSE>. Accessed: 2018-11-20. 7
- [101] ISO. ISO/IEC/IEEE 42010 Standard. <http://www.iso-architecture.org/ieee-1471/>. Accessed: 2018-11-20. 15
- [102] ISO. ISO/IEC/IEEE 42010 International Standard: Systems and Software Engineering: Architecture Description, 2011. 56, 77
- [103] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering, 2007. 22
- [104] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The orc programming language. In *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009.*, pages 1–25, 2009. 45
- [105] G. Kotonya and I. Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998. 9

- [106] M. Kovács, D. Varró, and L. Gönczy. Formal analysis of BPEL workflows with compensation by model checking. *Comput. Syst. Sci. Eng.*, 23(5), 2008. 45
- [107] O. Kovalenko, E. Serral, M. Sabou, F. J. Ekaputra, D. Winkler, and S. Biffl. Automating Cross-Disciplinary Defect Detection in Multi-Disciplinary Engineering Environments. In *Knowledge Engineering and Knowledge Management*, pages 238–249. Springer, 2014. 100
- [108] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004. 18
- [109] B. Lambeau, C. Damas, and A. van Lamsweerde. Process execution and enactment in medical environments. In *Software Engineering in Health Care*, pages 145–161. Springer, 2014. 19
- [110] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. 42
- [111] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979. 42
- [112] C. Lange, M. R. V. Chaudron, J. Muskens, L. J. Somers, and H. M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *Incompleteness of UML Designs, Proc. Workshop on Consistency Problems in UML-based Software Development, 6 th International Conference on Unified Modeling Language, UML 2003*, 2003. 42
- [113] J. Le Noir, O. Delande, D. Exertier, M. da Silva, and X. Blanc. Operation based model representation: Experiences on inconsistency detection. In R. France, J. Kuester, B. Bordbar, and R. Paige, editors, *Modelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 85–96. Springer Berlin Heidelberg, 2011. 30, 35, 38, 39
- [114] leankit. What is Kanban? <https://leankit.com/learn/kanban/what-is-kanban/>. Accessed: 2018-11-20. 9
- [115] E. Lee. The problem with threads. Technical report, University of California at Berkeley, 2006. 122
- [116] J. Li, Y. Fan, and M. Zhou. Performance Modeling and Analysis of Workflow. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 34(2):229–242, March 2004. 47
- [117] B. Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM. 11
- [118] R. Lopez-Herrejon and A. Egyed. Detecting inconsistencies in multi-view models with variability. In T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors, *Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science*, pages 217–232. Springer Berlin Heidelberg, 2010. 30, 39

- [119] R. E. Lopez-Herrejon and A. Egyed. Towards fixing inconsistencies in models with variability. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 93–100, New York, NY, USA, 2012. ACM. 30, 37, 39
- [120] F. J. Lucas, F. Molina, and A. Toval. A systematic review of uml model consistency management. *Inf. Softw. Technol.*, 51(12):1631–1645, Dec. 2009. 28, 29
- [121] L. Lucio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. FTG+PM: an integrated framework for investigating model transformation chains. In *SDL 2013: Model-Driven Dependability Engineering - 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*, pages 182–202, 2013. 4, 19, 63, 109
- [122] Y. Ma, G. Qi, P. Hitzler, and Z. Lin. *Measuring Inconsistency for Description Logics Based on Paraconsistent Semantics*, pages 30–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 42
- [123] MathWorks Inc. MATLAB API for Java. <https://www.mathworks.com/help/matlab/matlab-engine-api-for-java.html>. Acc.: 2016-08-02. 172
- [124] MathWorks Inc. Matlab Website. [mathworks.com/products/matlab](http://mathworks.com/products/matlab). Acc.: 2016-08-02. 162
- [125] MathWorks Inc. Multibody Modeling. <https://www.mathworks.com/help/physmod/sm/multibody-modeling.html>. Acc.: 2018-12-16. 150
- [126] MathWorks Inc. SimEvents Website. [mathworks.com/products/simevents](http://mathworks.com/products/simevents). Acc.: 2016-08-02. 88
- [127] T. Mens and R. Van Der Straeten. Incremental resolution of model inconsistencies. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *Recent Trends in Algebraic Development Techniques: 18th International Workshop, WADT 2006, La Roche en Ardenne, Belgium, June 1-3, 2006, Revised Selected Papers*, pages 111–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 139
- [128] T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer Berlin Heidelberg, 2006. 30, 33, 38, 39, 40, 104
- [129] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. 167
- [130] M. Minsky. Matter, mind and models. 1965. 11
- [131] T. Miu and P. Missier. Predicting the execution time of workflow activities based on their input features. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 64–72, Nov 2012. 47
- [132] modelpractice. General Model Theory by Stachowiak. <https://modelpractice.wordpress.com/2012/07/04/model-stachowiak/>. 11

- [133] P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004. 1
- [134] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 19
- [135] S. Mustafiz and H. Vangheluwe. Explicit modelling of statechart simulation environments. Proceedings of the Summer Simulation Multiconference, pages 21:1–21:8, 2013. 128
- [136] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 455–464, May 2003. 30, 39, 40
- [137] OASIS. Human Task (WS-HumanTask) Version 1.1. <http://docs.oasis-open.org/ns/bpel4people/ws-humantask/200803>. Acc: 2017-08-17. 19
- [138] OASIS. WS-BPEL 2.0 Specification. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>. Acc: 2017-08-17. 19
- [139] Object Management Group (OMG). SysML Website. <http://www.omg.org/spec/SysML/>. Accessed: 2018-11-20. 15
- [140] Object Management Group (OMG). BPMN 2.0 Specification. <http://www.bpmn.org/>, 2011. Accessed: 2018-05-06. 19
- [141] A. Oka, S. Yamamoto, and S. Isoda. Consistency management for software design information repository. In *Computing and Information, 1993. Proceedings ICCI '93., Fifth International Conference on*, pages 579–585, May 1993. 30, 31, 39
- [142] OSLC Community. OSLC - Open services for lifecycle collaboration core specification version 3.0. <http://open-services.net>, 2017. Accessed: 2018-11-20. 45
- [143] L. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering, ICSE '87*, pages 2–13. IEEE Computer Society Press, 1987. 19, 119
- [144] C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007. 45
- [145] Oxford Dictionaries. Definition of "process". <https://en.oxforddictionaries.com/definition/process>. Accessed: 2018-11-20. 17
- [146] G. Pahl and W. Beitz. *Engineering design: a systematic approach*. Springer Science & Business Media, 2013. 18
- [147] Z. Pap, I. Majzik, A. Pataricza, and A. Szegi. Methods of checking general safety criteria in uml statechart specifications. *RELIABILITY ENGINEERING & SYSTEM SAFETY*, 87:89 – 107, 2005. 128
- [148] M. Persson, M. Törngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil. A characterization of integrated multi-view mod-

- eling in the context of embedded and cyber-physical systems. In *Proceedings of the International Conference on Embedded Software*, 2013. 1, 2, 56, 58, 140
- [149] K. Petersen, C. Wohlin, and D. Baca. The waterfall model in large-scale development. In *International Conference on Product-Focused Software Process Improvement*, pages 386–400. Springer, 2009. 18
- [150] A. Qamar, J. Wikander, and C. During. A mechatronic design infrastructure integrating heterogeneous models. In *Mechatronics (ICM), 2011 IEEE International Conference on*, pages 212–217, April 2011. 2, 30, 33, 38, 39, 40
- [151] X. Qin. Delayed consistency model for distributed interactive systems with real-time continuous media. *Journal of Software*, 6(13):1029–1039, 2002. 43
- [152] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC ’14, pages 122–131, New York, NY, USA, 2014. ACM. 30, 36, 38, 39
- [153] I. Rath, A. Hegedus, and D. Varro. Derived features for emf by integrating advanced model queries. In *European Conference on Modelling Foundations and Applications*, pages 102–117. Springer, 2012. 158
- [154] I. Rath, A. Okros, and D. Varro. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software & Systems Modeling*, 9(4):453–471, Sep 2010. 179
- [155] A. Rauzy and Y. Dutuit. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering & System Safety*, 58(2):127 – 144, 1997. {ESREL} ’95. 38
- [156] J. Reineke and S. Tripakis. Basic problems in multi-view modeling. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 217–232. Springer Berlin Heidelberg, 2014. 1, 26
- [157] M. H. Romanycia and F. J. Pelletier. What is a heuristic? *Computational Intelligence*, 1(1):47–58, 1985. 51
- [158] G. Roşu and S. Bensalem. Allen linear (interval) temporal logic–translation to ltl and monitor synthesis. In *International Conference on Computer Aided Verification*, pages 263–277. Springer, 2006. 139, 180
- [159] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 18(3):217 – 238, 2012. 78, 101
- [160] D. Schmelter, J. Greenyer, and J. Holtmann. Toward learning realizable scenario-based, formal requirements specifications. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pages 372–378. IEEE, 2017. 10
- [161] B. Schätz, F. Hödlz, and T. Lundkvist. Design-space exploration through constraint-based model-transformation. In *Engineering of Computer Based Systems (ECBS)*,

- 2010 17th IEEE International Conference and Workshops on, pages 173–182. IEEE, 2010. 82
- [162] A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995. 32
- [163] Scrum Alliance. Learn About Scrum. <https://www.scrumalliance.org/learn-about-scrum>. Accessed: 2018-11-20. 9
- [164] B. Selic. A short catalogue of abstraction patterns for model-based software engineering. *Int. J. Software and Informatics*, 5(1-2):313–334, 2011. 58
- [165] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003. 124, 178
- [166] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010. 42
- [167] A. A. Shah, A. A. Kerzhner, D. Schaefer, and C. J. J. Paredis. Multi-view modeling to support embedded systems engineering in sysml. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-driven Engineering*, pages 580–601. Springer-Verlag, Berlin, Heidelberg, 2010. 30, 34, 39
- [168] Siemens PLM. AMESim Website. <https://www.plm.automation.siemens.com/en/products/lms/imagine-lab/amesim/index.shtml>. Accessed: 2018-11-20. 151, 172
- [169] B. Silver and B. Richard. *BPMN method and style*, volume 2. Cody-Cassidy Press Aptos, 2009. 45
- [170] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei. Instant and Incremental QVT Transformation for Runtime Models. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS’11, pages 273–288, Berlin, Heidelberg, 2011. Springer-Verlag. 171
- [171] D. J. Sorin, M. D. Hill, and D. A. Wood. *A primer on memory consistency and cache coherence*. Number 16 in Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011. 42
- [172] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001. 1, 25, 26, 27, 29, 31, 32, 37, 40
- [173] H. Stachowiak. General model theory. *Springer*, 1973. 11
- [174] V. Stiehl. *Process-Driven Applications with BPMN*. Springer, 2014. 45
- [175] V. Stolz. An integrated multi-view model evolution framework. *Innovations in Systems and Software Engineering*, 6(1-2):13–20, 2010. 30, 35, 39
- [176] SymPy Development Team. SymPy Website. <http://www.sympy.org/>. Acc: 2017-08-17. 92, 104, 172

- [177] E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *Electronic Communications of the EASST*, 29, 2010. 178
- [178] E. Syriani, H. Vangheluwe, and B. LaShomb. T-core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, 14(3):1215–1243, 2015. 178
- [179] TechSim Engineering s.r.o. LMS Virtual.Lab-Motion Brochure. [http://techsim.cz/wp-content/uploads/Brochure\\_LMS-Virtual.Lab-Motion.pdf](http://techsim.cz/wp-content/uploads/Brochure_LMS-Virtual.Lab-Motion.pdf). Acc: 2018-08-17. 150
- [180] The Jython Project. Jython Website. <http://www.jython.org/>. Accessed: 2018-11-20. 172
- [181] The Open Group. The ArchiMate® Enterprise Architecture Modeling Language. <http://www.opengroup.org/subjectareas/enterprise/archimate-overview>. Accessed: 2018-11-20. 15
- [182] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC ’99, pages 163–172, New York, NY, USA, 1999. ACM. 43
- [183] A. Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2), 2004. 178
- [184] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1:80 – 99, 2015. 83, 178
- [185] W. van der Aalst. Business process management as the "killer app" for petri nets. *Software and System Modeling*, 14(2):685–691, 2015. 45
- [186] W. van der Aalst and A. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005. 19, 45
- [187] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. 19, 45, 46, 106, 107, 110, 119, 166
- [188] W. Van Der Aalst, K. M. Van Hee, and K. van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004. 17
- [189] W. M. Van der Aalst. Process discovery: An introduction. In *Process Mining*, pages 125–156. Springer, 2011. 179
- [190] R. Van Der Straeten. *Inconsistency Management in Model-Driven Engineering: An Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, Software Languages Lab, 2005. 29
- [191] R. Van Der Straeten and M. D’Hondt. Model refactorings through rule-based inconsistency resolution. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC ’06, pages 1210–1217, New York, NY, USA, 2006. ACM. 30, 36, 39

- [192] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between uml models. In P. Stevens, J. Whittle, and G. Booch, editors, «*UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2003. 30, 36, 39, 40
- [193] B. F. Van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and W. M. Van Der Aalst. The prom framework: A new era in process mining tool support. In *International conference on application and theory of petri nets*, pages 444–454. Springer, 2005. 179
- [194] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001. 10
- [195] S. Van Mierlo, Y. Van Tendeloo, I. Dávid, B. Meyers, A. Gebremichael, and H. Vangheluwe. A multi-paradigm approach for modelling service interactions in model-driven engineering processes. In *Proceedings of Mod4Sim*, Mod4Sim, part of the Spring Simulation Multi-Conference, pages 565–576, 2018. 113, 119
- [196] S. Van Mierlo, Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe. SCCD: SCXML extended with class diagrams. In *3rd Workshop on Engineering Interactive Systems with SCXML*, 2016. 46, 120
- [197] Y. Van Tendeloo. Foundations of a multi-paradigm modelling tool. In *MoDELS ACM Student Research Competition*, pages 52–57, 2015. 121
- [198] Y. Van Tendeloo and H. Vangheluwe. An overview of PythonPDEVS. In C. W. RED, editor, *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*, pages 59 – 66. Éditions Cépaduès, Apr. 2016. 113, 121
- [199] Y. Van Tendeloo and H. Vangheluwe. Classic DEVS modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, pages 644 – 656. IEEE, Dec. 2017. 107
- [200] Y. Van Tendeloo and H. Vangheluwe. The Modelverse: a tool for multi-paradigm modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, pages 944 – 955. IEEE, Dec. 2017. 66, 107, 113, 121, 172, 180
- [201] Y. Van Tendeloo and H. Vangheluwe. Extending the DEVS formalism with initialization information. *ArXiv e-prints*, 2018. 107
- [202] H. Vangheluwe. An introduction to multiparadigm modelling and simulation. In *Proceedings of the AIS'2002 Conference 9–20 (2002)*. 35. Press, 2000. 12, 13, 173, 175, 177
- [203] K. Vanherpen, J. Denil, I. Dávid, P. D. Meulenaere, P. J. Mosterman, M. Törngren, A. Qamar, and H. Vangheluwe. Ontological reasoning for consistency in the design of cyber-physical systems. In *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*, pages 1–8, April 2016. 14, 68, 80, 91, 100, 101, 133, 135, 136, 154, 168
- [204] Vanherpen, Ken. A contract-based approach for multi-viewpoint consistency in the concurrent design of cyber-physical systems. *University Antwerp*, 2018. 158, 179

- [205] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental Model Synchronization for Efficient Run-Time Monitoring. In S. Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *LNCS*, pages 124–139, 2009. 171
- [206] M. von Detten, C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story diagrams-syntax and semantics. *Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Tech. Rep. tr-ri-12-324*, 2012. 46
- [207] G. Wachsmuth. Modelling the operational semantics of domain-specific modelling languages. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 506–520. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 138
- [208] R. Wagner, H. Giese, and U. Nickel. A plug-in for flexible and incremental consistency management. In *Proc. of the International Conference on the Unified Modeling Language*, page 93, 2003. 1
- [209] G. G. Wang and S. Shan. Review of metamodeling techniques in support of engineering design optimization. *Journal of Mechanical design*, 129(4):370–380, 2007. 82
- [210] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005. 45
- [211] M. Weske. Business process management architectures. In *Business Process Management*, pages 333–371. Springer, 2012. 18, 177
- [212] J. M. Wilson. Gantt charts: A centenary appreciation. *European Journal of Operational Research*, 149(2):430–437, 2003. 19
- [213] Y. Xia, Y. Liu, J. Liu, and Q. Zhu. Modeling and performance evaluation of BPEL processes: A stochastic-petri-net-based approach. *IEEE Trans. Systems, Man, and Cybernetics, Part A*, 42(2):503–510, 2012. 45
- [214] Z.-j. Xiao, H.-y. Chang, and Y. Yi. Method of workflow time performance analysis. *COMPUTER INTEGRATED MANUFACTURING SYSTEMS-BEIJING-*, 12(8):1284, 2006. 47
- [215] Ximeng Sun. A Model-Driven Approach to Scenario-Based Requirements Engineering. [http://msdl.cs.mcgill.ca/people/simon/publications/thesis\\_singlesided.pdf](http://msdl.cs.mcgill.ca/people/simon/publications/thesis_singlesided.pdf). 10
- [216] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000. 12, 107, 120