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

Universiteit Antwerpen

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

To my Father.

Contents

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 Vangheluwe.
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 together 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 are 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 did not achieve world dominance while I was working with you, I can promise that I'll carry the torch with great responsibility, 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 years: Yentl, Joachim, Simon, Ken, Bart, Claudio and Ali. Your presence created the most inspiring professional environment I've ever got the chance to work in.
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: Vasco Amaral, Antonio Cicchetti, Joachim Denil, Serge Demeyer, 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, Dani and Laci.

Last but not least, I would like to extend my gratitude to a special person who stood by me as long as she could. Thank you, Barbara, for having been my rock during these trying times. Thank you for having been the companion throughout this long 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 engineered systems has increased drastically over the past decades. Pertinent examples are mechatronic and cyber-physical systems, characterized by an enormous complexity, stemming from the number and heterogeneity of the components and concerns involved in their engineering.

Due to this complexity, ensuring the correctness of the system is a challenging task.

Model-based systems engineering (MBSE) proposes modeling the system, before it gets realized. Multi-paradigm modeling (MPM), specifically, advocates modeling every aspect of the system at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). In such settings, the system components are developed in parallel, enabling a more efficient engineering. Parallelism, however, gives rise to inconsistencies between the design artifacts, compromising the ultimate correctness of the system.

In this work, we argue, that managing inconsistencies in an MBSE setting is an effective heuristic for managing the ultimate correctness of the system.

First, we formulate an appropriate definition for correctness and consistency in terms of semantic properties. Then, we define a process-based approach to provide formal foundations for the detection and management of inconsistencies. The process model is transformed so that the potential inconsistencies are managed, and its transit time is minimal. This is achieved by searching through the set of potential process candidates by the techniques of multi-objective design space exploration. The exploration is guided by the transit time performance metric of the process, obtained via a DEVS-based simulation. A general mapping between process models and DEVS provided.

The utility of the approach is shown through a demonstrator, using the prototype tooling developed to support the approach.

Netherlandstalige Samenvatting

De complexiteit van de systemen die we bouwen is de afgelopen decennia drastisch toegenomen. Relevante voorbeelden zijn mechatronische en cyberfysische systemen, gekenmerkt door een enorme complexiteit, die voortvloeit uit het aantal en de heterogeniteit van de componenten.

Vanwege deze complexiteit is het garanderen dat een systeem correct ontworpen wordt een grote uitdaging.

Modelgebaseerd systeemontwerp (Model Based Systems Engineering - MBSE) stelt voor het systeem expliciet te modelleren voordat het wordt gerealiseerd. Multi-paradigma modellering (MPM) pleit met name voor het modelleren van elk aspect van het systeem op de meest geschikte niveau(s) van abstractie, gebruik makend van de meest geschikte formalisme (s), en met expliciet modelleren van de complexe ontwikkelingsprocessen. Vaak worden de systeemcomponenten door meerdere ingenieurs parallel ontwikkeld. Parallelisme leidt echter mogelijks tot inconsistenties tussen de ontwerpartefacten, waardoor de ultieme correctheid van het systeem in het gedrang komt.

In dit werk argumenteren we dat het beheersen van inconsistenties tussen systeemcompenten/gezichtspunten in een MBSE-omgeving een effectieve heuristiek is voor het bekomen van ultieme correctheid van het hele systeem.

Eerst formuleren we een geschikte definitie voor correctheid en consistentie in termen van semantische eigenschappen. Dan definiëren we een procesgebaseerde benadering om een formele basis te bieden voor de detectie en het beheer van inconsistenties. Het procesmodel van het systeemontwerp wordt getransformeerd zodat de potentiële inconsistenties worden beheerd terwijl de ontwikkeltijd van het systeem minimaal is. Dit wordt bereikt door de verzameling van potentiële proceskandidaten met multi-objectieve ontwerpruimte exploratie te doorzoeken. De verkenning wordt geleid door de transittijdprestatiemetriek van het proces, verkregen via een op DEVS gebaseerde simulatie.

Het nut van de aanpak wordt aangetoond door middel van een demonstrator: het ontwerp van een Automated Guided Vehicle. Dit gebeurt met behulp van de prototype tooling die ter ondersteuning werd ontwikkeld.

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
- Coference 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

List of Tables

List of Listings

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

Ever since the first industrial revolution at the end of the 18th century, the number of engineered systems has been growing at an ever increasing pace. More then two centuries later, the fundamental challenge to any engineering endeavor remains the same: delivering the *correct product*. The correct system is the one that meets the functional and the extra-functional requirements formulated against the system.

The *complexity* of engineered systems has been steadily increasing in past centuries. With the advent of mechatronic (Figure 1.1) and cyber-physical systems (CPS), and movements such as the Internet-of-Things (IoT), System-of-systems (SoS) and the Industry4.0, the complexity of systems exhibited an even more drastic increase. This is due the inherent heterogeneity of nowadays' systems. [151].

Ensuring the correctness of nowadays' highly complex systems is a serious engineering challenge.

Model-based systems engineering (MBSE) advocates *modeling* the system before it gets realized. This way, the various properties of the eventual system can be computed beforehand, resulting in an improved system design and decreasing costs. Due to the complexity, however, 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 points of view on the system. Multi-paradigm modeling (MPM) [134] acknowledges this idea and proposes modeling every aspect of the system explicitly, using the most appropriate formalisms, at the most appropriate levels of abstraction.

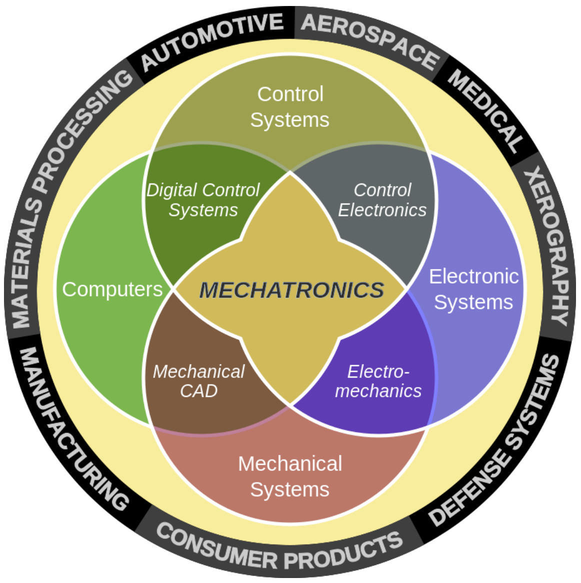


Figure 1.1: Mechatronic design requires an interplay between disparate domains. [41]

Adhering to MPM principles results in (i) *parallelized engineering processes*, in which the components of the system are developed independently; and (ii) stakeholders accessing the same system component through *multiple disparate views*.

Such settings give rise to potential *inconsistencies* between the views/models of the different stakeholders.

Overlaps in the semantic domain of models have been identified as the primary reason of model inconsistencies by many authors [151, 90, 215]. That is, properties of different models often turn out to be logically connected or sometimes even (nearly) the same [71, 82]. 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 exacerbates the problem. In Chapter 3, we further elaborate on the state of the art of inconsistency management.

The presence and absence of inconsistencies has strong implications to the eventual correctness of the product.

Finkelstein [70] suggests that “*rather than thinking about removing inconsistency we need to think about managing consistency*”. Managing (ensuring) consistency, however, does not necessary result in an eventually correct product. On the other hand, however, inconsistent views on the same system will surely deteriorate the engineering endeavor from delivering the correct product. That is, *inconsistency implies incorrectness*.

In this work, we claim that instead of managing *consistency*, one should *inconsistency*. In Chapter 4, we further elaborate on this idea.

Managing inconsistency still does not guarantee a correct product, but there is a reasonable expectation, that the majority of eventually incorrect system configurations can be avoided. In this sense, *managing inconsistency is a heuristic to the correct product*[161].

There are two fundamental types of techniques to managing inconsistency. *Preventive* techniques, ensure no inconsistencies can happen at any point of the engineering endeavor. *Resolution-based* techniques are more reactive in the sense, that inconsistencies are allowed to surface, but subsequently are required to be detected and resolved. The various techniques are reviewed in Chapter 3.

As of which technique to employ, at which point of the engineering endeavor, and in what scope, is a configuration specific to the given case. The selection, however, will have a profound impact on the overall efficiency of the engineering endeavor. To reason about the impacts of the selection, the *engineering process* is required to be modeled explicitly.

The modeled process can be augmented with inconsistency management activities and simulated for performance indicators (PI), such as transit time and costs. This way, different process alternatives can be evaluated to decide the most optimal one. The optimal process alternative is the one that *minimizes the number of inconsistencies* as much as possible, while keeping the *transit time and the costs low*. Other PIs can be selected for specific engineering settings, such as maximal resource utilization, minimal queueing time, etc.

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 an appropriate formalism and 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 work, in Section 8.1 we will answer the questions in greater details, linking them to the contributions, presented below.

Contributions

The contributions of this work are enumerated below. Next to each contribution is listed the section of this work that explicitly describes it.

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

Contribution 2: A new definition of model (in)consistency, in terms of semantic properties. Such an approach enables explicit reasoning about model inconsistencies potentially hidden in the semantic domain of models. (Chapter 4)

Contribution 3: A process modeling formalism, serving as the foundation for the rest of the contributions. The formalism supports linking engineering activities and artifacts with semantic properties. (Section 5.1)

Contribution 4: A technique for the off-line management of inconsistencies, in order to transform the original engineering process into a managed one. At the same time, the efficiency of the process is ensured by minimizing the transit time of the process. This is achieved by means of rule-based multi-objective design space exploration. (Section 5.2)

Contribution 5: Enactment of the optimized process, in order to enable the management of inconsistencies potentially appearing during the enactment. The enactment is achieved by explicitly modeled model transformations. (Section 5.3)

Contribution 6: On-line management of inconsistencies in order to manage inconsistencies appearing during the enactment. Symbolic mathematics are used for the continuous evaluation of artithemtic and first-order-logic constraints. (Section 5.4)

Contribution 7: DEVS-based simulation of processes for evaluating the performance of the process candidates in the off-line management phase. A mapping library between general process patterns to DEVS is provided, along with a simulation algorithm for computing the transit time. (Section 5.5)

Contribution 8: External service integration in order to enable using domain-specific engineering tools in the modeled process. The integration is achieved by explicitly modeled communication protocols, using the SCCD formalism. (Section 5.6)

Approach

In this work, we propose an approach for inconsistency management in the engineering of complex, heterogeneous systems. The approach consists of six contributions, shown in Figure 1.2. Most of the contributions provide automated machinery for carrying out the specific steps of the approach. The automation is achieved by an appropriate tooling, presented later in this work. The only exception, naturally, is the process modeling step, which still has to be carried out manually.

Structure

The rest of this work is structured as follows. Chapter 2 provides the necessary background information on the techniques and foundations relevant to this work. Chapter 3 reviews the state of the art (SOTA) related to this work. 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 work 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 shows the utility of the previously presented techniques by applying them in a demonstrative 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 work and points towards possible directions for future work.

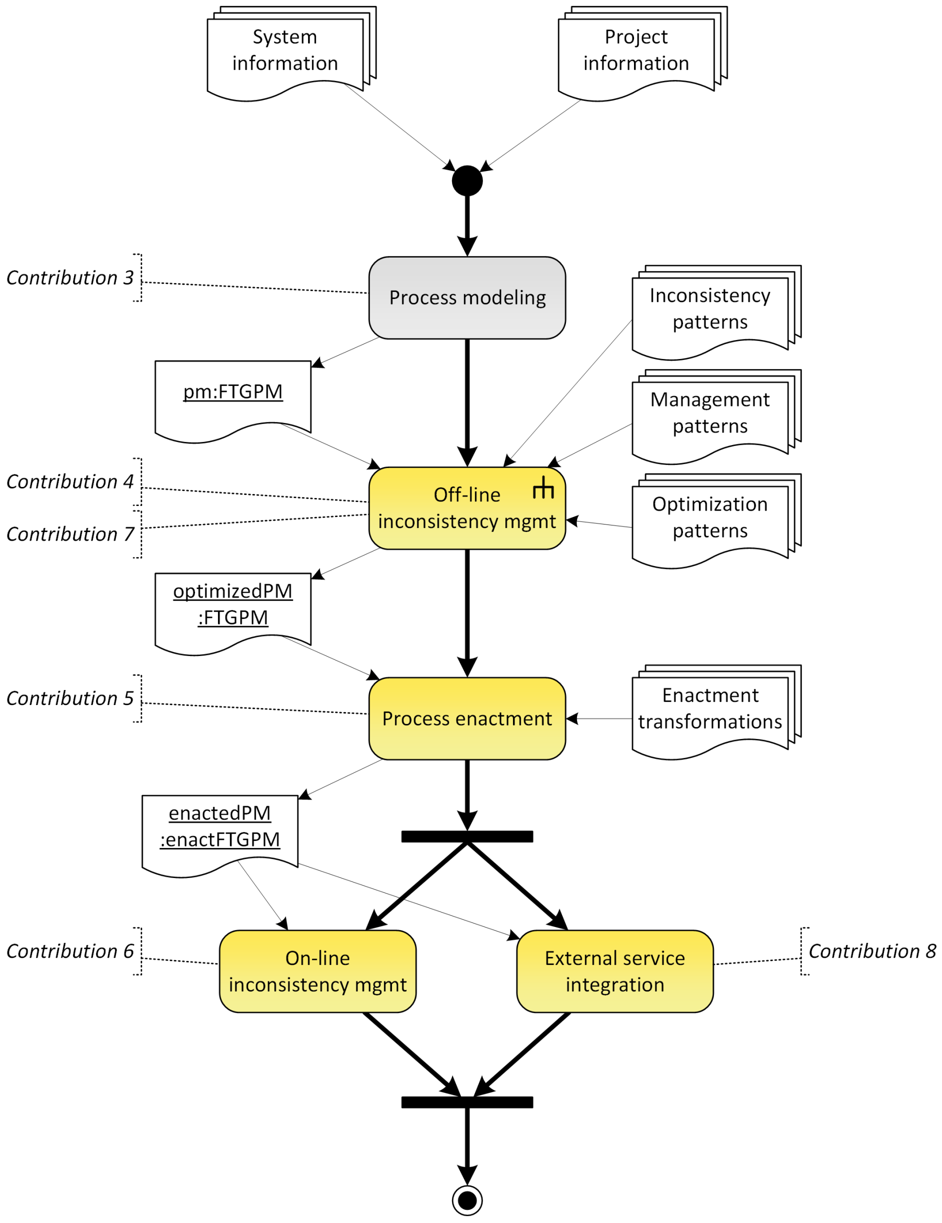


Figure 1.2: The process of the inconsistency management approach.

Chapter 2

Background

In this chapter, we give an overview on the foundational concepts related to the broader scope of this work. 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) [64] 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*.

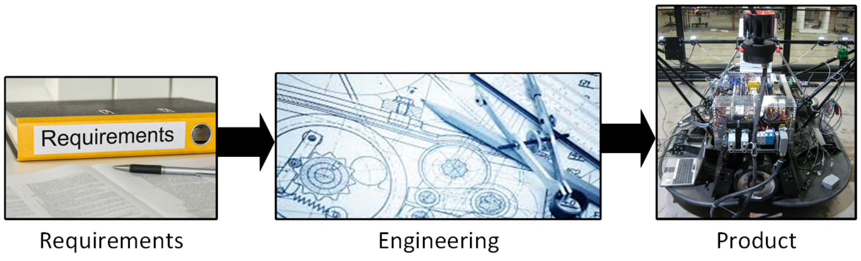


Figure 2.1: General overview of the engineering process.

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

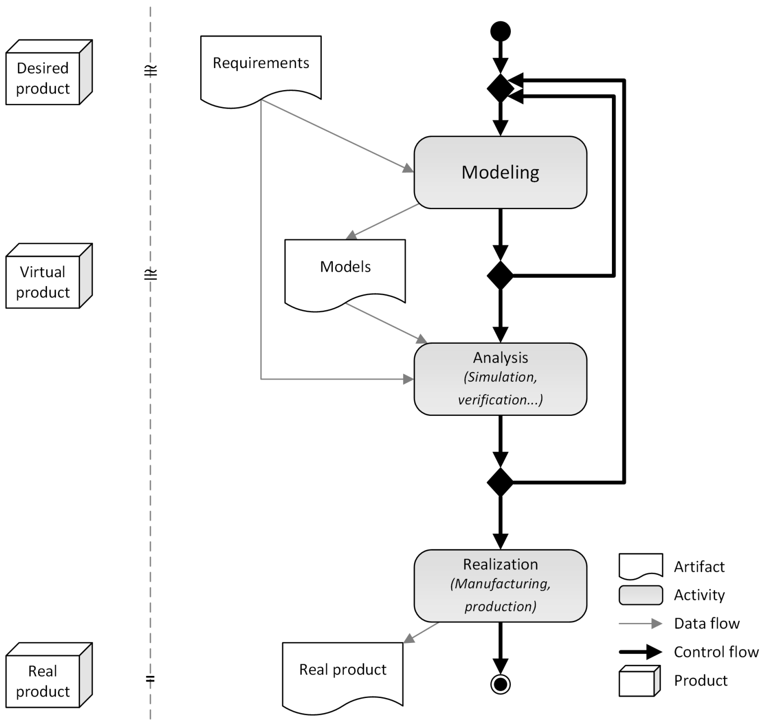


Figure 2.2: Model-based engineering.

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. According to the more formal definition by the CMMI standard [37], 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 [168] 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 [53].

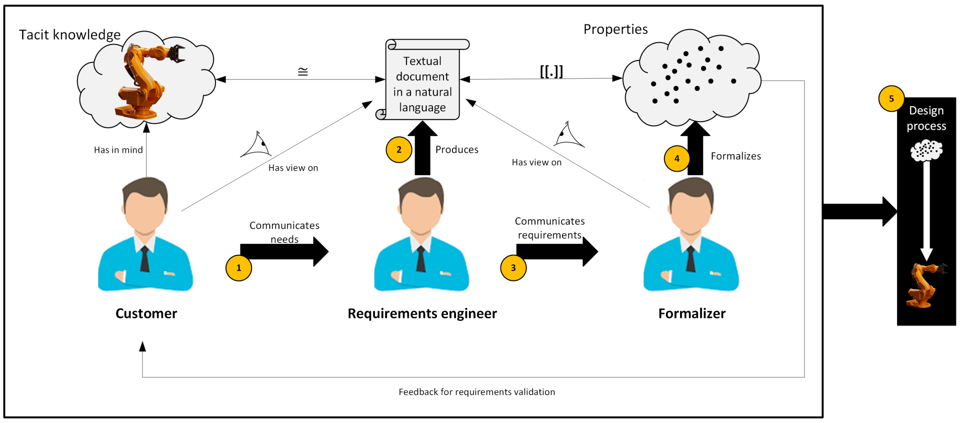


Figure 2.3: The typical workflow of requirements engineering.

Requirements engineering [105] aims at the proper specification and formalization of the customer needs. Figure 2.3 shows the four typical steps of requirement engineering: (i) communication of the tacit knowledge between the Customer and the Requirements engineer; (ii) externalization of the tacit knowledge in the form of textual requirements artifacts by the Requirements Engineer; (iii) communication of the externalized requirements between the Requirements Engineer and the Formalizer; and (iv) expressing the requirements as properties to be satisfied by the final system.

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 [39] 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 understating for the human stakeholders. Typical approaches include scenario-based [165] and goal-oriented requirements engineering [201], such as KAOS [44], I* [222] and GBRAM [9].

In a subsequent step, the requirements may be formalized. The task of the formalizer is translating the requirements to properties, by 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 [182].

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

In the remainder of this work, we assume that requirements can be correctly and efficiently translated into properties.

2.1.2 Virtual product

Before the actual system or product gets realized, the engineers are concerned with its architecture, parameter and behavior with respect to 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 [92, 27]. Atkinson and Draheim [12] refer to the VP as the single underlying model (SUM).

In this work, 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, 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 [131] (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* [118] 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 [179]. (Translation given by [133].)

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 [32].

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* [23]. 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.

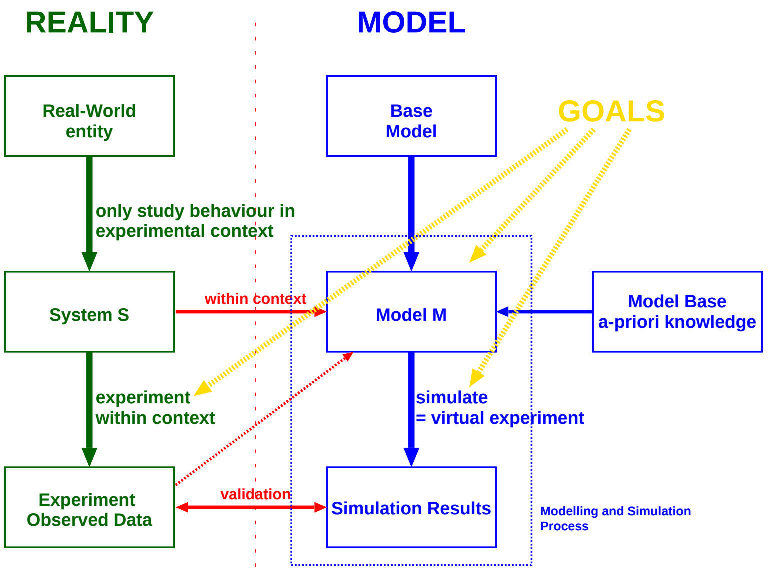

REALITY
MODEL
Real-World entity
Base Model
GOALS
only study behaviour in experimental context
System S
within context
Model M
Model Base a-priori knowledge
experiment within context
simulate = virtual experiment
Experiment Observed Data
validation
Simulation Results
Modelling and Simulation Process

Figure 2.4: Modeling and simulation concepts [210].

Figure 2.4 presents the concepts of modeling and simulation, as introduced by Vangheluwe et al. [210] and Zeigler [223]. 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 behavior (within a certain accuracy). The *System* is a well defined object in the *Real World* under specific conditions, only considering specific aspects of its behavior. *Experimental Frames* describe a limited set of circumstances under which a system is to be observed or subjected to experimentation. The purpose of building the *Model* is to enable virtual experimentation with the *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 [210] suggest embracing the principles of multi-paradigm modeling (MPM).

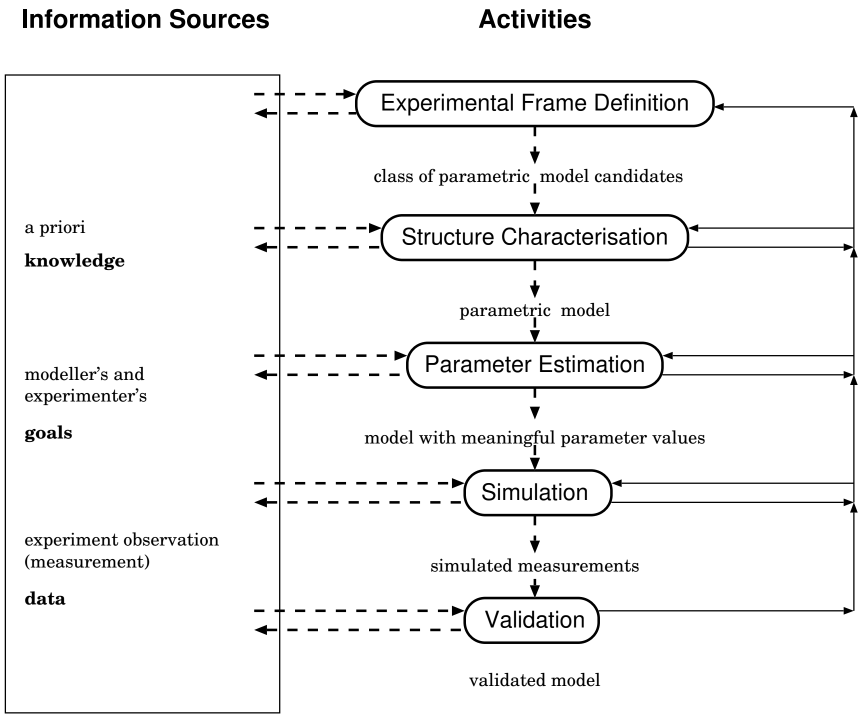


Figure 2.5: Model-based systems analysis [210].

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 work, 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.

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 work, 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 alone. Consider how UML [142] considers property merely a named “structural feature”; while some object-oriented languages (such as C#) consider class members with a purpose between a field and a method a property [130]; and at the same time properties bear the notion of the abstract representation of a real-world phenomena in an ontological, set theoretical way in the works of Vanherpen [211], Vangheluwe [210], and others. 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 weather 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. If a proof exists that a property can not be 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 work, 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: $0 < m < 10$ [kg].

Property-based approaches have been shown to be extremely feasible in detecting inconsistency and incorrectness in the engineering of highly complex heterogeneous systems [212].

In Chapter 4, we definition correctness and consistency in terms 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.

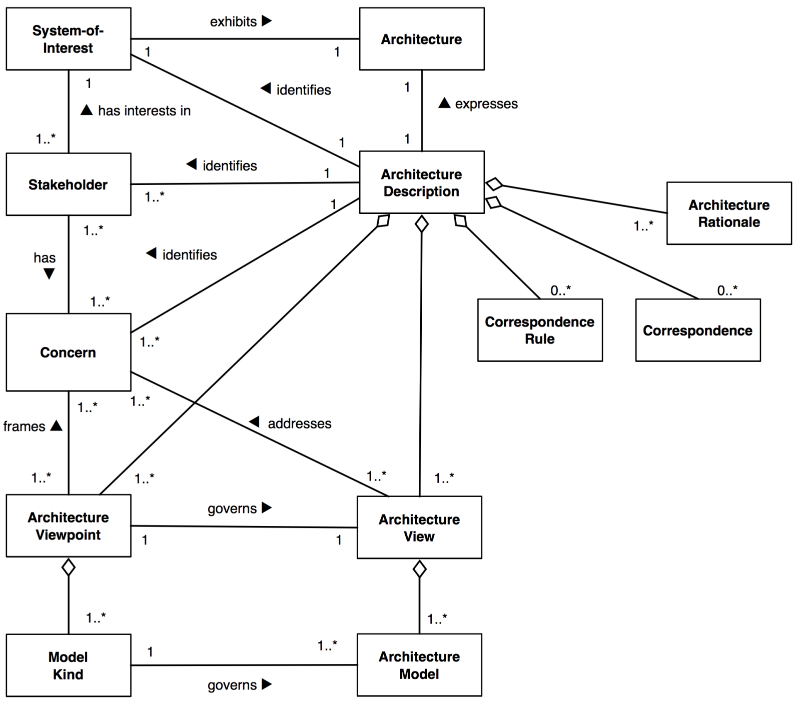


Figure 2.6: The Core of Architecture Description in ISO/IEC/IEEE 42010:2011 [101].

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 *Stakeholder*s will form the eventual *Architecture Viewpoint*s. Applying an Architecture Viewpoint 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 [188] and SysML [141].

Broman et al. [33] 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. [40] 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

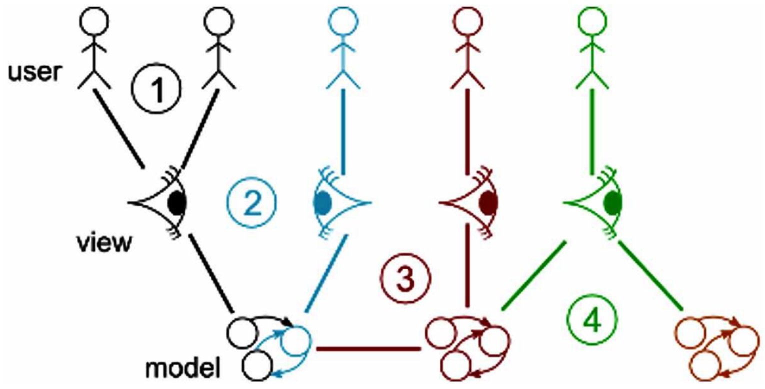


Figure 2.7: Scenarios in multi-view modeling [40].

Corley et al. [40] 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 throught 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 typically requires a 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. For example, in the mechanical design of complex systems, multi-body models and more detailed finite element models are often 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, 82]. 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, etc. The conformance of the system to such properties can be evaluated by simulations or model checking.

To reason about the characteristics of the semantic domain of models (such as overlaps and linked properties), an explicit representation of the semantic domain is required.

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 [148] 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 [195]. 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

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 [218],

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 [6], 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 [218]). 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. [149] 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 [153], the Rational Unified Process (RUP) [108] gained widespread visibility all around the software domain. Later on, to overcome the inherit rigidity of these process models, the processes have been shifted towards more of the iterative and incremental nature [30]. Enabled by the mechanisms of MBSE, the V-model and the subsequent Y-model have been popularized in safety-critical systems engineering [36]. Modern day software methodologies often build on the concept of *agile* [13] 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 work 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 [143] and BPEL [140]. These process models are often enactable, 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 [122], Activity diagrams [54], YAWL [193], MWE2 [59] or Apache Maven [10]. 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 [135], Statecharts [84], scheduling models (e.g. Gantt-charts [219]), 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 stat 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 [146, 194, 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 [139].

Chapter 3

State of the art

In this chapter, we present **Contribution 1** of this work: a mapping study of the state of the art. The majority of this section has been published in [55].

The chapter is organized into two main sections: Section 3.1 discusses the state of the art in (in)consistency management, while Section 3.2 discusses the state of the art in process engineering. Both areas have been thoroughly researched by multiple authors of multiple domains, constituting a vast body of knowledge. We have conducted a mapping study on (in)consistency management, keeping our focus on the applicability and scalability of the techniques in in heterogeneous system engineering settings.

Systematic mapping is a methodology that is frequently used in medical research, but that has largely been neglected in software engineering [152]. The usual outcome is a visual map classifying the results. It requires less effort than a systematic literature review while providing a more coarse-grained overview.

3.1 (In)consistency management

In this section, we review the state of the art on (in)consistency management. A mapping study has been carried out at the beginning of this research to understand the various aspects of the field. At this point, 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?**

The rest of this this section is structured as follows. In Section 3.1.1, the review process is discussed. In Section 3.1.2, we give an overview of our findings. In Section 3.1.3, we discuss the various features, patterns and activities typically encountered in inconsistency management. In Sections 3.1.4–3.1.6, the typical patterns of the (in)consistency management activities of characterization, detection and resolution are discussed, respectively. In Section 3.1.7, the optional and auxiliary aspects of (in)consistency management are discussed. In Section 3.1.8, the conclusions are drawn.

3.1.1 Process

Our process for producing the mapping study discussed in this section was based on the guidelines described in [152] and [103]. The process is shown in Figure 3.1.

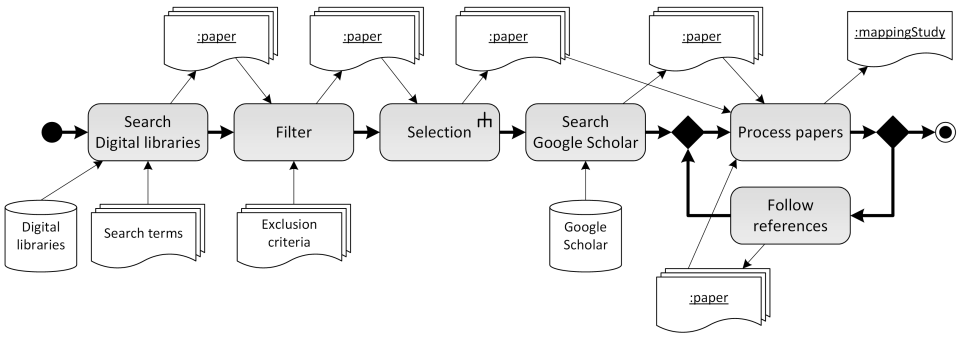


Figure 3.1: Process of producing the current mapping study.

Search Digital libraries

The first step to the process is the search for the relevant scientific papers. For this, the appropriate *Search terms* were determined, along with the relevant *Digital libraries*. The Cartesian product of these two input parameterized the *Search* step; i.e., every search terms has been evaluated against every digital library.

The following search terms were selected:

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

Additionally, the following extra search terms were also used:

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

- mechatronic design;
- concurrent engineering;
- tool integration.

The following digital libraries were used:

- ACM Digital Library [4];
- IEEE Xplore Digital Library [96];
- Springer Link [178].

For every search term, the hit list in every library has been ordered based on *relevance*, which is a compound metric of impact, number of citations, recency, and a similarity metric between the search term and the content of the paper. The top 15 papers for every search term have been accepted as candidates for the further evaluation. 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 number of unique papers included for the fist round was approximately 200.

Filter

Two exclusion criteria were defined to filter the selected papers based on: (i) non-English papers, and (ii) short papers got rejected at this point in the process.

Selection

In the *Selection* step, the set of initial 200 papers have been narrowed down, and the set of papers to be processed have been selected.

Two reviewers have been working on the selection step; each of them following one instance of the detailed process of the *Selection* activity, shown in Figure 3.2.

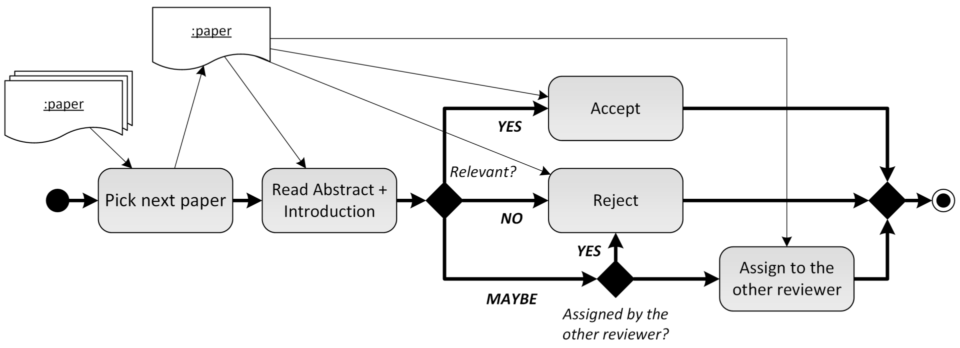

:paper
:paper
Pick next paper
Read Abstract + Introduction
Relevant?
YES
NO
MAYBE
Accept
Reject
YES
Assigned by the other reviewer?
Assign to the other reviewer

Figure 3.2: The detailed process of the *Selection* activity.

The set of initial 200 papers have been formed into a queue. In the fist step, the reviewer *chose the next paper* for investigation.

After *reading through the abstract and introduction*, the relevance of the paper has been determined, and as a result, the paper was assigned a status out of the following three:

- Accept – the paper was found relevant, and selected for processing;
- Reject – the paper was found not relevant, and was not selected for processing;
- Maybe – the relevance of the paper could not be determined.

If the relevance of the paper could not be determined (*Maybe*), the paper was assigned to the other reviewer to repeat the process. If the second reviewer could not clearly accept the paper, it got rejected.

The *relevance* of the papers has been determined based on the following criteria. A paper fulfilling any of the below criteria got immediately assigned the *Accepted* status.

- Presents an overview.
  - It's a survey or systematic review.
  - Presents a classification of the types of (in)consistency problems.
  - Scoped to the domain of mechatronics, CPS, avionics, automotive.
- Presents a technique
  - for overlap detection.
  - for inconsistency detection.
  - for resolution.
  - the technique is supported via a (prototype) tooling.

Papers not meeting any of the above criteria, but otherwise interesting and potentially useful, got assigned the *Maybe* status.

The rest of the papers got assigned the *Rejected* status.

Search Google Scholar

Google Scholar (`https://scholar.google.be/`) is one of the most popular scientific indexing libraries. Therefore, a search on Google Scholar has been additionally carried out. Duplicate entries have not been introduced. A set of approximately 25 papers have been identified for further processing.

Process papers

Eventually, approximately 70 papers have been identified for being processed and incorporated into the mapping study.

During the processing, the list of citations of each paper has been processed as well, and the relevant references have been followed.

Eventually, 59 papers on the topic of (in)consistency management have been incorporated into the mapping study. Some of them contribute to the vocabulary and taxonomy provided later in this section; some of the contributed to understanding the mechanisms and techniques of the specific steps in (in)consistency management.

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. In software modeling, and especially in the case of UML modeling, inconsistencies are typically defined 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 domains of concurrent engineering, which is not linked directly to MPM, but we had *a-priori* knowledge about potential links.

In the rest of Section 3.1, we discuss the findings of our study.

3.1.2 Overview

Here, we give a brief overview of our findings.

Definitions

There is no unified definition to "inconsistency", as it is always specific to 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. [177] 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. [89] also add that

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

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

We find the latter definition to be the best suited for domain of heterogeneous systems engineering out of the existing definitions. The definition, however, does not consider inconsistencies that remain hidden in the semantic domain [210] of models. To this end, we suggest a more appropriate definition of inconsistency, in terms of *semantic properties*. In our terms, two design artifacts d_i and d_j are said to be consistent w.r.t. the set of properties $P' \equiv P_{req}(d_i) \bigcap P_{req}(d_j)$ iff $\forall p \in P' : d_i \vDash p \Leftrightarrow d_j \vDash p$. That is, properties that are in the overlap of the respective semantic domains of d_i and d_j, are either have to be jointly satisfied by the d_i and d_j, or jointly not satisfied, in order to consider d_i and d_j consistent. If d_i and d_j are not consistent, they are inconsistent. The definition is explained in great details in Section 4.3.1.

The sources of inconsistencies

Investigating their sources, gives a better understanding on the nature of inconsistencies. Huzar et al. [95] note that the **imprecise or vague semantics** of modeling languages (and UML, specifically) being a potential source of inconsistencies:

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. [177]

Views conforming to these viewpoints are highly interrelated due to the concerns addressed overlapping. These interrelations and overlaps can lead to inconsistencies. [89]

(. . .) 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? [160]

Additionally, the **evolution** of single model artifacts can also lead to inconsistencies. Hehenberger et al. [87] 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.*

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 **in**consistencies, since we assume that inconsistencies cannot be avoided and the real challenge is to deal with them *after* they are encountered.

3.1.3 Features and patterns in inconsistency management techniques

In this section we define a structured taxonomy for discussing various inconsistency management techniques and approaches. In our taxonomy, we combine those of Spanoudakis et al. [177] and Hanmer et al. [83] and characterize inconsistency management techniques and by their *features* and the *patterns* used for implementing those features. (For example, using *Pivot models* (Section 3.1.4.1) is a typical pattern of the *Representation* feature (Section 3.1.4.1).) 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.

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).

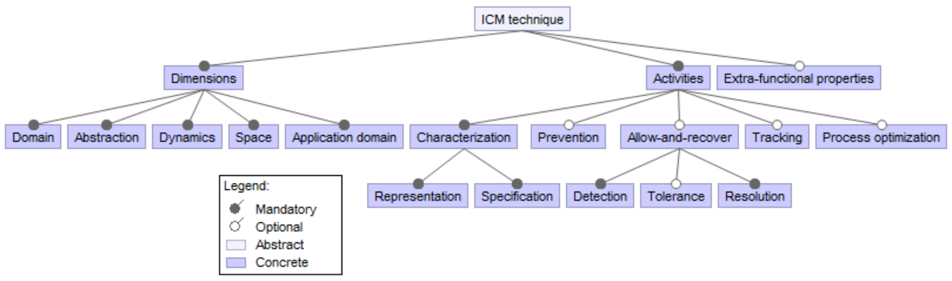


Figure 3.3: Feature model of the inconsistency management techniques.

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

Table 3.1: Features of inconsistency management and their interpretation.

3.1.3.1 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 [121]. 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 (ĩnter-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. [121] 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.

3.1.3.2 Activities

Activities define *how* an inconsistency management process is structured. We mainly follow the classification of Spanoudakis et al. [177], but further extend it as presented later. Our classification of inconsistency management processes is shown in Figure 3.4.

Characterization

Characterizing inconsistencies is the 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. [177] do not emphasize the role of proper inconsistency characterization, while Van Der Straeten [197] acknowledges characterization as the foundational first step towards a managed approach towards handling inconsistencies.

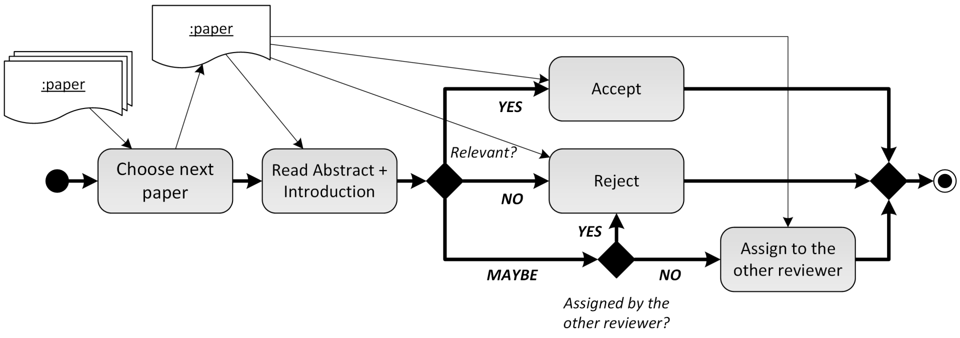

:paper
:paper
Choose next paper
Read Abstract + Introduction
Relevant?
YES
NO
MAYBE
Accept
Reject
YES
NO
Assigned by the other reviewer?
Assign to the other reviewer

Figure 3.4: The process of managing an inconsistency.

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. Finkelstein et al. [70] note, however, that managing consistency does not only mean removing the detected inconsistencies; but also *preserving* inconsistency where it is desirable.

3.1.3.3 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.

	Characterization	**Detection**	**Resolution**	**Prevention**
Adourian [5]	●	●	●	○
Balzer [14]	●	●	○	○
Becker [18]	●	○	○	○
Bhave [26]	●	●	○	○
Bhave 2 [25]	●	○	○	○
Blanc [29]	●	●	○	○
Dahman [43]	●	●	●	○
Easterbrook [56]	●	●	○	○
Egyed [61]	●	●	○	○
Engels [63]	●	●	○	○
Gausemeier [76]	●	●	●	○
Giese [79]	●	●	●	○
Giese 2 [78]	○	○	○	●
Hamlaoui [62]	●	●	○	○
Hehenberger [87]	●	●	○	○
Herzig [88]	●	○	○	○
Hessellund [91]	●	○	○	●
Le Noire [113]	●	●	○	○
Lopez-Herrejon [119]	○	●	○	○
Lopez-Herrejon 2 [120]	●	●	◐	○
Mens [128]	●	●	●	○
Nentwich [137]	○	○	●	○
Oka [144]	●	○	○	●
Qamar [154]	●	●	○	○
Quinton [156]	○	●	○	○
Shah [172]	●	○	○	○
Stolz [181]	●	●	○	○
Van Der Straeten [199]	●	●	●	○
Van Der Straeten 2 [198]	●	●	●	○

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

3.1.4 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.

3.1.4.1 Representational patterns

Naming conventions

According to Spanoudakis [177]:

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. [91] 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. [144] 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 to 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 [177] 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 employ a specific subset of dependency models in form of TGGs [167].

Becker et al. [18] 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 (bidirectionalaty is preferred) and adaptability (support for different domains) among others and identify TGGs as a good fit with these requirements.

Adourian et al. [5] 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. [76] 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 the conceptual design, all views of the system 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. [79] focus on bidirectional model synchronization as an important technique in MDSD. They provide an algorithm based on the triple graph grammar formalism (TGG). Their previous work [80] optimizes for a single change, which is generalized to compound changes in [79]. The problem of multiple transformation steps is identified, along with the need for a formal framework for reasoning about algebraic-temporal structures of these steps.

Mechatronic/CPS settings.

Qamar et al. [154] 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).

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 [61] 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. [43] 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. [62] 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. [172] 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. [25] 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. [26] 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. [29] 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 and 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. [181] 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. [87] 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. [198] 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. [156] 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. [63] 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. [78] 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. [199] 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. [120] pose the research question "where should the fixes be placed?" rather then 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.

3.1.4.2 Specification patterns

The most common approach to specify inconsistency rules is to manually inspect models and make links between the appropriate model elements. Spanoudakis [177] 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. [88] present 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.

3.1.5 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* [177]. In general, the patterns of characterization heavily influence the operational patterns of detection.

Graph reasoning

Techniques employing dependency graphs pattern typically rely on the related techniques of graph reasoning, such as graph querying, transformations, pattern matching, etc. Hamlaoui et al. [62] use EMFCompare [58] to enrich the correspondence model by change deltas among models. Giese et al. [79] 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. [154] and Adourian et al. [5] 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. [18]. 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 solver-based detection, typically using a variant of Prolog for tooling. Hessellund et al. [91] employ this patten in combination with naming conventions, 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. [29] and Le Noire et al. [113] use Prolog in combination with operation-based representation. In the former case, edit operations are recorded and traces are fed to a Prolog engine, resulting in an on-demand, batch style inconsistency check approach. The latter approach builds on SWI-Prolog. No difference in the time and space complexity of the detection phase in operation-based and standard representation patterns has been identified. Quinton et al. [156] use SAT solvers to evaluate sequences of edit operations in operation-based representation scenarios. Model changes are translated to the BR4CP/ Aralia language [159] 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 Hehenberger et al. [87] could employ it.

Incremental evaluation

The incremental evaluation pattern is one of the typical requirements for inconsistency management techniques described by Becker et al. [18]. Some authors explicitly aim to exploit the benefits of such an approach (Egyed et al. [61], Giese et al. [79], Gausemeier et al. [76]). While not always emphasized, incremental evaluation is usually handled as a desirable property of inconsistency management techniques.

In Table 3.4, we summarize the characterization and detection patterns supported by the inconsistency management techniques investigated up to this point.

	Characterization					**Detection**
	Representation			**Specification**		
	Naming conventions	**Dependency graphs**	**Other**	**Human inspection**	**Similarity analysis**	**Incremental**
Adourian [5]	○	●	○	●	○	◐
Balzer [14]	○	○	○	●	○	◐
Becker [18]	○	●	○	●	○	◐
Bhave [26]	○	●	○	●	○	◐
Bhave 2 [25]	○	○	● (pivot)	●	○	*N/A*
Blanc [29]	○	○	● (operation)	●	○	●
Dahman [43]	○	●	○	●	○	◐
Easterbrook [56]	○	○	● (logic)	●	○	○
Egyed [61]	○	●	○	●	○	●
Engels [63]	○	○	● (formal)	●	○	○
Gausemeier [76]	○	●	○	●	○	●
Giese [79]	○	●	○	●	○	●
Giese 2 [78]	○	○	● (formal)	○	○	●
Hamlaoui [62]	○	●	○	●	○	●
Hehenberger [87]	○	○	● (ontology)	●	○	●
Herzig [88]	○	●	○	○	●	*N/A*
Hessellund [91]	●	○	○	●	○	◐
Le Noire [113]	○	○	● (operation)	●	○	◐
Lopez-Herrejon [119]	N/A (avoidance only)					◐
Lopez-Herrejon 2 [120]	○	○	● (logic)	●	○	●
Mens [128]	○	●	○	●	○	◐
Nentwich [137]	N/A (avoidance only)					◐
Oka [144]	●	○	○	●	○	◐
Qamar [154]	○	●	○	●	○	◐
Quinton [156]	○	○	● (SAT)	●	○	○
Shah [172]	○	○	● (pivot)	●	○	○
Stolz [181]	○	○	● (operation)	●	○	●
Van Der Straeten [199]	○	○	● (logic)	●	○	○
Van Der Straeten 2 [198]	○	○	● (rules)	●	○	○

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

3.1.6 Resolution patterns

Resolution patterns address detected inconsistencies. Depending on their intent, changing and non-changing actions are distinguished [177], 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. [76] 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 [5, 43, 76, 79, 128, 137, 154]. 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 3.1.5).)

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. [91] 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. [85] 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. [48].

Logic-based resolution

The logic-based representational pattern (Section 3.1.4.1) is often used in conjunction with similarly logic-based resolution patterns, such as in Van der Straeten et al. [199].

3.1.7 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.

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. [29] 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. [14] 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. [56] 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.

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.

Measuring inconsistency

Measuring inconsistency has been a topic of interest in the ontology and knowledge engineering. Hunter and Konieczny [93] 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 [123] 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 [15], 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.

3.1.8 Conclusions

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

3.1.8.1 Surveying questions

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. This is due to the likely scalability issues when the complexity of the underlying engineered system increases drastically, as it is the case in CPS/mechatronic systems. The number and heterogeneity of the components and stakeholder views in such systems pose a severe issue w.r.t. the scalability of inconsistency management techniques.

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 (Hesselund et al. [91], Hegedus et al. [85]), the appropriate resolution actions typically require complex sequences of changes. Explicitly modeling resolution actions is a missing link in the state of the art.

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.

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. As opposed to this, 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.

3.1.8.2 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 [171]; Sorin et al.'s book provides a good overview [176] 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 [31]. 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* [21], while the latter can be further broken down into perceptive (or absolute) consistency [31], where every process executes events at the same absolute time, *delayed consistency* [155], which imposes a fixed, maximum pair-wise delay of $\delta(i, j)$ between processes i and j, and *timed consistency* [189], which requires a fixed global bound of δ 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.

3.2 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 [194]. Pertinent examples include languages based on the *Business Process Model and Notation* [180], *Petri nets* [192] and *UML Activity Diagrams* [24].

The *Business Process Model and Notation* (BPMN) [174] 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 [42] 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) [217] 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. [147] and Xia et al. [220]. Kovács et al[106] use a symbolic analysis model checker. Fu at al. [75] 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) [193] 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) [145] 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 [84] 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 [203], to provide structural object-oriented language constructs (*i.e.*, objects with behaviour).

Story diagrams [214] 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, event 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**
Petri nets	●	○	●	●	◐
Activity Diagrams	●	○	●	●	●
BPMN2.0	●	○	●	●	●
jBPM	●	●	●	○	●
BPEL	●	◐	●	●	◐
YAWL	●	○	●	●	●
Orc	●	◐	●	●	○
OSGi	○	◐	●	○	●
OSLC	○	●	●	○	●
FTG+PM	●	○	●	●	●
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. [8] 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 [194], which is paramount to this work. 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 [38]. 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. [221] 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. [132] 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) [11] 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.

3.2.3 Design structure matrices

Design structure matrices (DSM) [65] are widely used in mechanical engineering for modeling *structures* of processes, by capturing the dependencies between the activities of a process.

DSM are adjacency matrices with the activities in the two dimensions. The control flow between the activities is not made explicit, but it is implicitly encoded in the ordering of the activities.

A number in the i-th column and j-th row represents the fact of activity i requiring data/information from activity j. Depending on the semantics of the specific DSM implementation, this number can be a binary 0/1 information; or, typically, a weight of dependency, ranging from 0 to 3.

DSM are primarily used as the structured data for algorithms employed in process improvement scenarios. Example approaches include [34, 77, 117], which document the reduced number of iterations required in the improved process. The operations typically used in such algorithms include:

Partitioning. Minimizes the number of activities executed multiple times due a dependency. In a DSM, numbers below the main diagonal represent information flow to an activity that has already been performed. The partitioning step identifies these cases and attempts to reorder the activities accordingly.

Clustering. Organizes the engineering teams enacting the different activities. This is achieved by grouping the various activities into clusters with the minimal amount of information flows between them. [163]

Tearing. Breaks dependency loops by identifying the optimal breaking point with the optimal sequence of activities.

The approach presented in this work cannot be fully aligned with DSM in terms of the operations used for process improvement/optimization. This is due to the fact that DSM use operations with local optimization heuristics, as opposed to the approach approach presented in this work, which relies on global optimization heuristics. The global optimization heuristics of the approach described in this work are: (i) having every potential inconsistency managed; and (ii) minimal transit time of the process. DSM operations use partitioning, clustering and tearing in order to support the minimal transit time global heuristic. The approach described in this work does not demand these local optima to be satisfied, as the global optimum may differ from such configurations.

The approach presented in this work, however, is capable of emulating DSM, and offers techniques beyond the standard ones used in DSM. Some of the notable examples are the following.

- DSM capture the level/strength of dependency between activities by an estimate range value. (Typically 0-3.) By explicitly modeling artifact flows in our processes, and introducing the property model with system characteristics and levels of influence, the approach presented in this work enables reasoning about the dependencies in processes in a more detailed way.
- DSM focus solely on data/information flow, and are missing the control flow of the process. At the same time, DSM do not provide enactment semantics. The approach presented in this work provides execution semantics.
- The semantics of DSM are not sufficient for quantitative performance analysis of the underlying process. In this work, we show, that explicit process models can be efficiently translated to DEVS models for performance evaluation.

Chapter 4

Correctness and consistency

In this section, we present **Contribution 2** of this work: 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 work 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

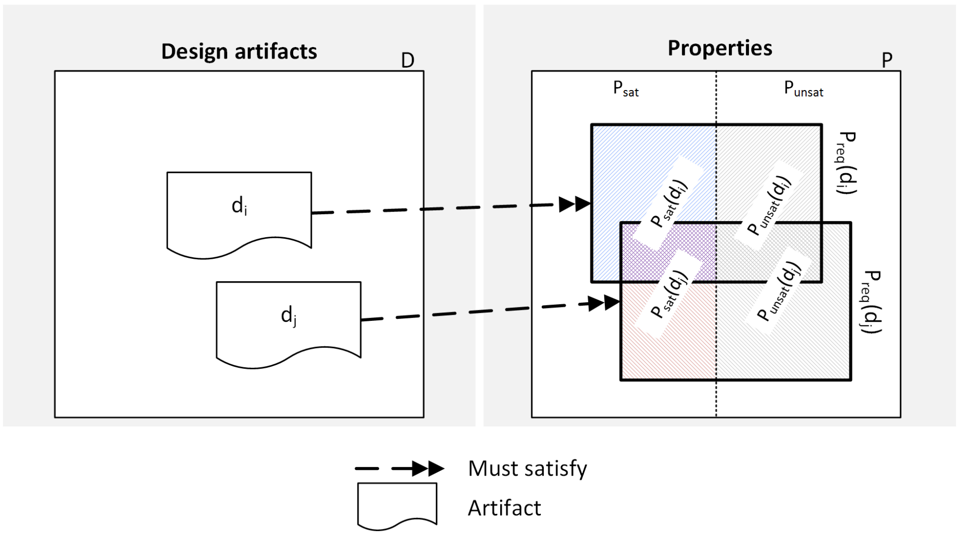


Figure 4.1: The relationship between properties and designs.

The set of system properties P can be divided as follows. $P_{req}(d_i) \subseteq P$ 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 $P_{sat}(d_i) \subseteq P$ and $P_{unsat}(d_i) \subseteq P$, respectively. Formally, the following hold:

- $\forall p \in P_{sat}(d_i) : d_i \vDash p$;
- $\forall p \in P_{unsat}(d_i) : d_i \nvDash p$;
- $P_{sat}(d_i) \bigcup P_{unsat}(d_i) \equiv P_{req}(d_i)$.

Note, that $P_{req}(d_i) \not\equiv P$, meaning $P_{sat}(d_i) \bigcup P_{unsat}(d_i) \not\equiv P$. This convention is followed to allow open world semantics. That is, $\forall p \in P \setminus P_{req}(d_i)$ the satisfaction is allowed to be unknown.

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 work 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 work. 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 $\forall p \in P_{req}(d_i) : d_i \vDash p$.
As a consequence, there must be no unsatisfied required property: $P_{unsat}(d_i) \equiv \emptyset$.

The correctness of the overall system is given by the correctness of the virtual product. The correctness of the virtual product, in turn is given by the correctness of the design artifacts.

Correctness of the system.
The system $\mathcal{S}$, given by the virtual product $\mathcal{S} = \bigcup_{d_i \subseteq D} d_i$ is said to be correct with respect to $P' \subseteq P$ iff $\forall p \in P' \; \exists d_i \in D : d_i \vDash p$.

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.

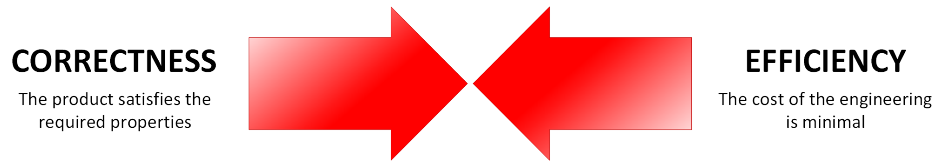

CORRECTNESS
The product satisfies the required properties
EFFICIENCY
The cost of the engineering is minimal

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 [161]). 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 work, 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 are more likely to lead to a correct product is indeed plausible. At the same time, this notion also warns us about the fact that consistent design *does 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. the set of properties $P' \equiv P_{req}(d_i) \bigcap P_{req}(d_j)$ iff $\forall p \in P' : d_i \vDash p \Leftrightarrow d_j \vDash p$.

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*. $(P_{req}(d_i) \bigcap P_{req}(d_j))$ 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.

	$d_i \vDash P'$	$d_j \vDash P'$	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. $P' \equiv P_{req}(d_i) \bigcap P_{req}(d_j)$.

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:

- $consistent(d_i, d_j) \not\Rightarrow correct(\mathcal{S})$;
- $correct(\mathcal{S}) \Rightarrow consistent(d_i, d_j)$.

Conversely:

- $inconsistent(d_i, d_j) \Rightarrow incorrect(\mathcal{S})$;
- $incorrect(\mathcal{S}) \not\Rightarrow inconsistent(d_i, d_j)$.

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.

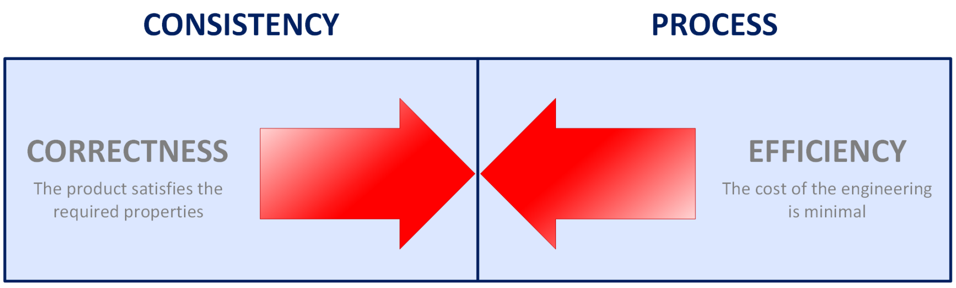

CONSISTENCY
PROCESS
CORRECTNESS
The product satisfies the required properties
EFFICIENCY
The cost of the engineering is minimal

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

In the later sections of this work, 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*.

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. [19] propose that three elementary operators (and their combinations) are suitable for describing the activities of engineers in contract-based engineering settings:

- architectural decomposition;
- view decomposition;
- abstraction followed by refinement.

Such mechanisms, indeed, are well suited to describe engineering activities, and consequently, engineering processes. The authors use contracts and system components as the primitives to the above operations. We elaborate on the above operations in terms of *properties* to give an overview as of how properties used in our approach emerge.

We also introduce an additional typical engineering step: *refinement followed by abstraction*, which is inverted direction of the last operator out of the above ones. To allow flexibility to this list of operations, we also allow a wildcard: the *elementary engineering operation*. The concept is shown in Figure 4.4.

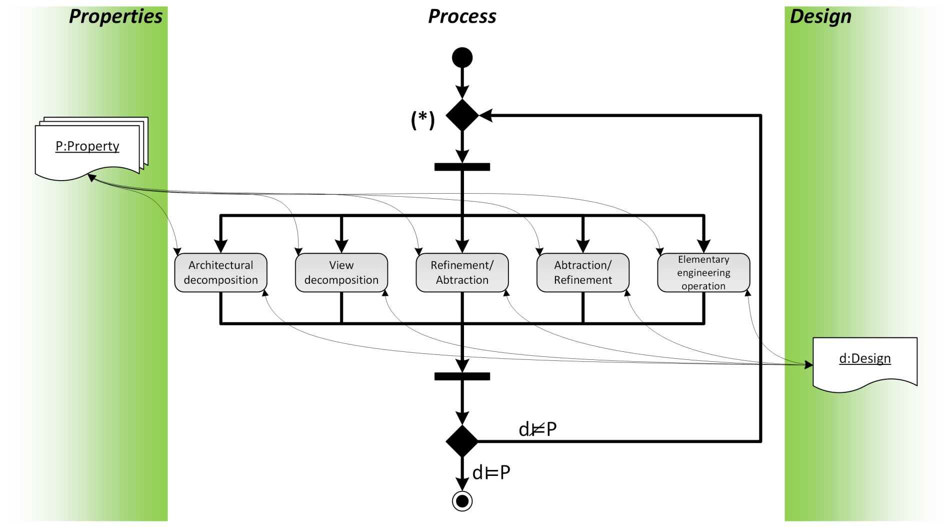


Figure 4.4: The five types of engineering activities.

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.

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.

A typical example in the AGV case is breaking down the system into a mechanical and electrical subsystem.

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.

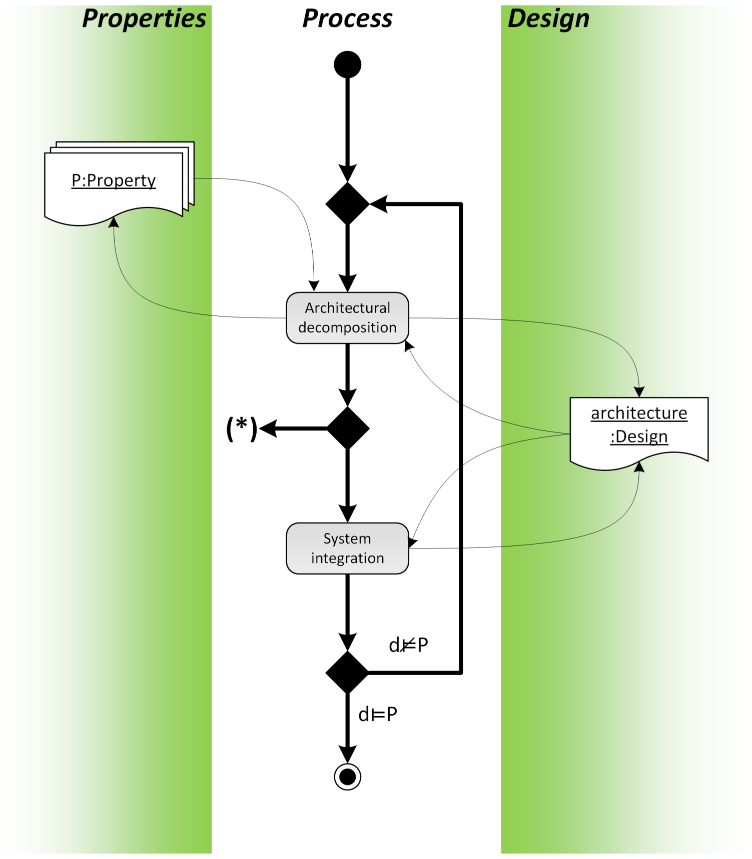


Figure 4.5: The sub-process of *Architectural decomposition* and *System integration*.

This sub-process also appears in the work of Benveniste et al. [19], as part of the contract composition and system integration scenarios.

4.5.2 View decomposition

The view decomposition activity (Figure 4.6) 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 [33]. Applying a viewpoint to a specific problem results in a view.

From our perspective, the decomposition the system into various views is a correspondence

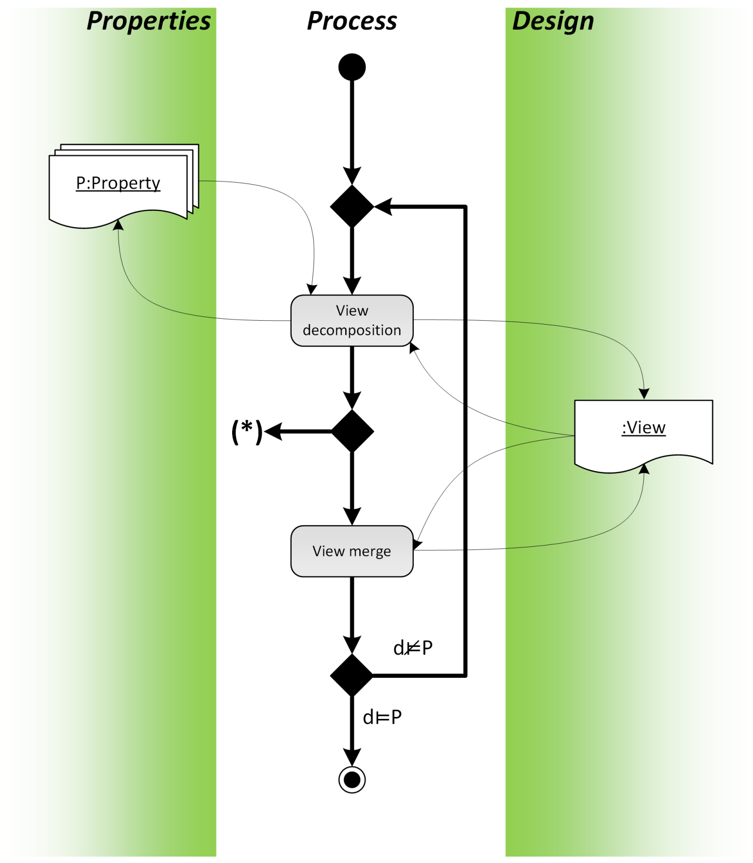


Figure 4.6: The sub-process of *View decomposition* and *View merge*.

function which projects a subset of properties: $decompose_v(sys) : v \mapsto P' \subseteq P$. 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 [151] and gives rise to inconsistencies as the engineering process 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 $v' \models P' = \{p'_1, p'_2 \ldots p'_M\}$ and $v'' \models P'' = \{p''_1, p''_2 \ldots p''_N\}$, it must hold that $v' \oplus v'' \models P' \bigcup P''$, i.e. $v' \oplus v'' \models p'_1 \wedge p'_2 \wedge \ldots \wedge p'_M \wedge p''_1 \wedge p''_2 \wedge \ldots \wedge p''_N$. This step is used when two views have to be jointly investigated, e.g., for co-simulation.

A typical example in the AGV case is when the mechanical engineer's and the electrical engineer's views on the motor and battery are created. In the mechanical view, the battery is characterized by its mass and shape; while in the electrical view, it is the capacity that is represented. Similar concerns apply for the motor. The two views are eventually merged in the electormechanical view.

4.5.3 Abstraction-refinement

Abstraction and refinement are natural mechanisms in engineering processes.

As shown in Figure 4.7, the *Abstraction* activity takes a design d with its related properties P, and produces another design d' and its related properties P'. The activity is an abstraction iff $P' \subseteq P$, i.e. the number of concerned properties decreases. The abstraction is valid iff $\forall p \in P : d \vDash p \Rightarrow d' \vDash p$, i.e. the properties originally satisfied by the design d must be satisfied by the design d'.

A typical example in the AGV case is when the control engineer first abstracts the system to be controlled as an inertia matrix, and later refines it into an actual plant model of the system.

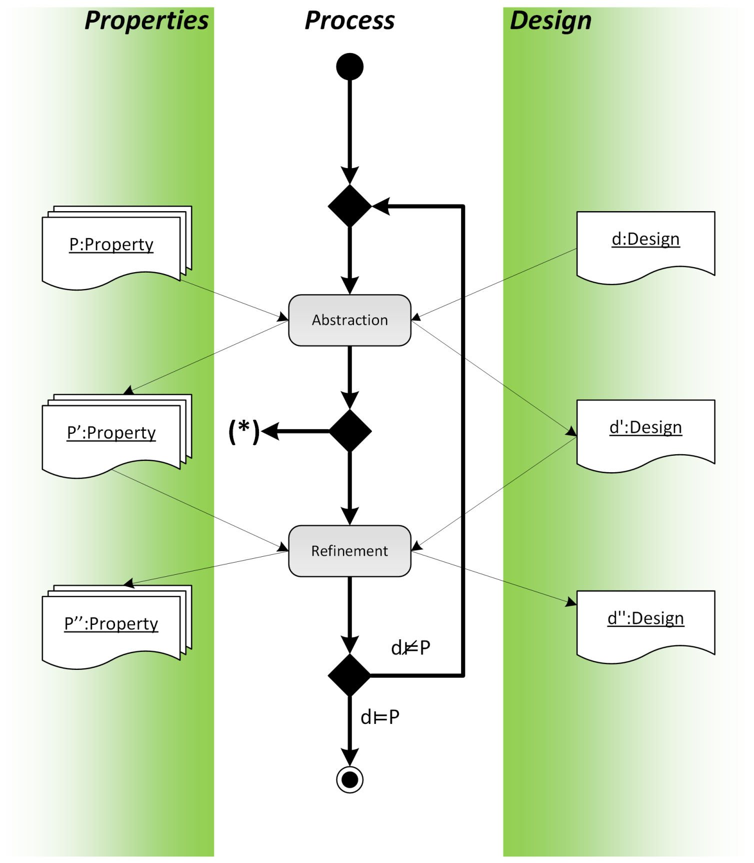


Figure 4.7: The sub-process of *Abstraction-Refinement*.

Persson et al. [151] 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 [169] provides a catalogue of abstraction patterns for model-based software engineering. 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 an abstraction iff $P' \subseteq P''$, i.e., the number of concerned properties increases. The refinement is valid iff $\forall p' \in P' : d' \vDash p' \Rightarrow d'' \vDash p''$, i.e., the properties originally satisfied by the design d' must be satisfied by the design d''.

4.5.4 Refinement-abstraction

As the inverted direction of the abstraction-refinement mechanism, the refinement-abstraction mechanism (Figure 4.8) first refines a model a then abstracts it. We have noticed this mechanism being frequently used in industrial practice, in particular, in mechanical engineering. A pertinent example is when a finite element model (FEM) is constructed first, then the model is refined into a rigid body dynamics model; and subsequently, the model is abstracted into a finite state machine (FSM) for model checking purposes.

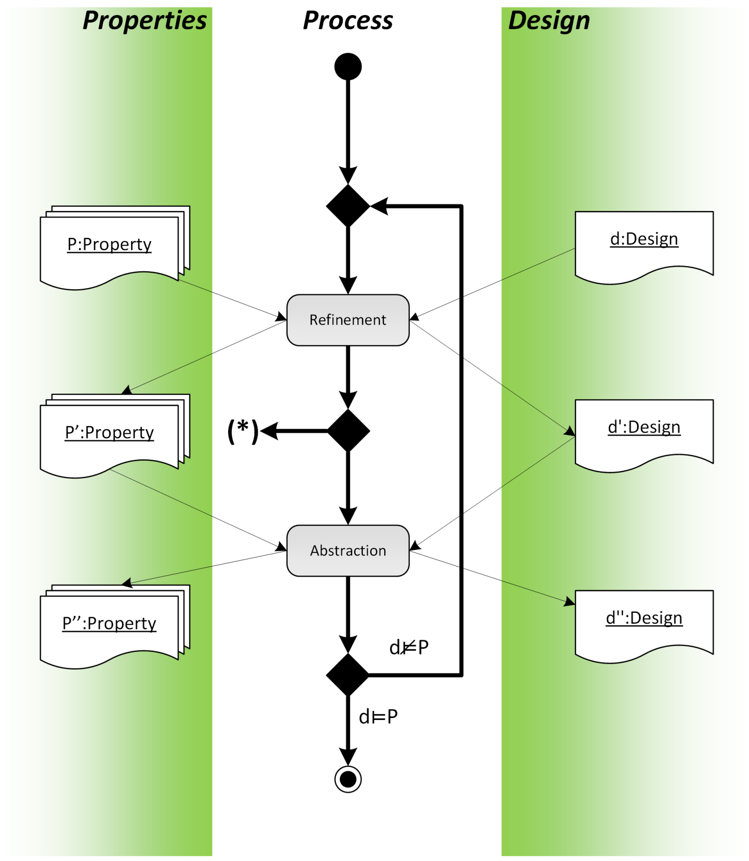


Figure 4.8: The sub-process of *Refinement-Abstraction*.

4.5.5 Elementary engineering operation

The elementary engineering operation (Figure 4.9) 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 formulated against the engineered system.

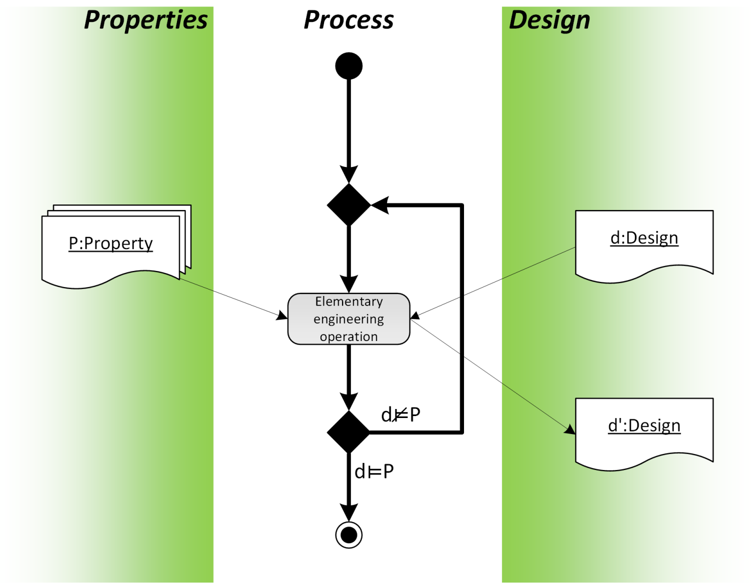


Figure 4.9: The sub-process of the *Elementary engineering operation*.

Chapter 5

Process-oriented inconsistency management

This chapter constitutes the core of this work, and presents the inconsistency manage framework and methodology developed in 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.

An illustrative case

We use various elements of a real-live case study of the design of an automated guided vehicle (AGV) in this chapter [22]. 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

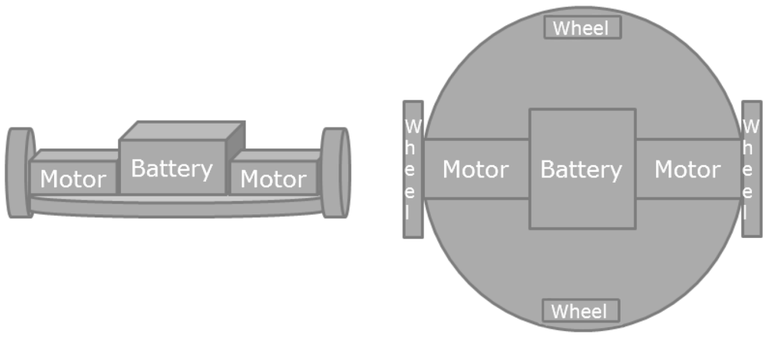


Figure 5.1: Front and top view of the conceptual design of the AGV.

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.

We consider the AGV to be representative for the types of systems our approach aims to support to the design of. This is due to the AGV exhibiting the typical complexity mechatronic and cyber-physical systems exhibit. This complexity is due to the number and heterogeneity of the involved components, concerns and views.

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.

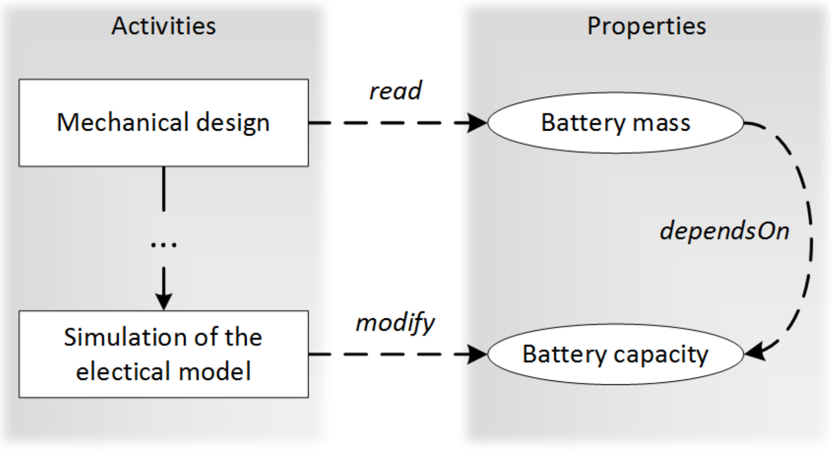


Figure 5.2: Running example.

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 work: a formalism for modeling engineering processes. The majority of this section has been published in [45] and [46].

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[122] 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.

Additionally, we extend the FTG+PM formalism by allowing explicit modeling of (i) properties, and (ii) costs. 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.

We kept the modeling language visual, but extended it with textual elements. Process models, however, tend to scale up quickly, and the graphical notation only works if appro-

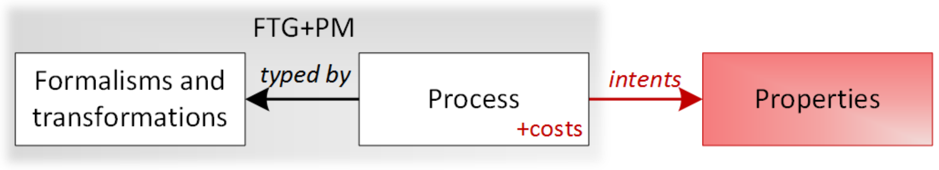


Figure 5.3: Relationships between processes, formalisms and properties.

priate packaging mechanisms are provided for grouping various concerns. In our prototype, packaging is not provided, but layering mechanisms help alleviating this issue.

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 we mean a partially ordered set of activities A. The process is defined by the tuple $\langle A, \Delta_c, D, \Delta_d \rangle$ consisting of

- the set of activities A;
- the set of directed control relations between activities $\Delta_c : A \rightarrow A$;
- the set of artifacts D;
- the set of directed data flow relations between activities and artifacts $\Delta_d : A \rightarrow D$;

5.1.1.2 Transitivity of the control flow

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

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

5.1.1.3 The formalism-transformation graph (FTG)

The FTG consists of formalisms and transformations, formally: $FTG = \langle \mathcal{F}, \mathcal{T} \rangle$. A transformation transforms formalisms into formalisms: $t \in \mathcal{T} : \mathcal{F} \rightarrow \mathcal{F}$.

The FTG serves as a type system to processes, where formalisms type the artifacts of the process; and transformations type the activities of the process.

- $\forall\, d \in D\; \exists\, f \in \mathcal{F}$: typeOf(d) = f.

- $\forall\, a \in A\, \exists\, t \in \mathcal{T}$: typeOf(a) = t.

In the running 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, $a_{MechDesign} \in A$, and typeOf($a_{MechDesign}$)$\in \mathcal{T}$.

5.1.1.4 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.

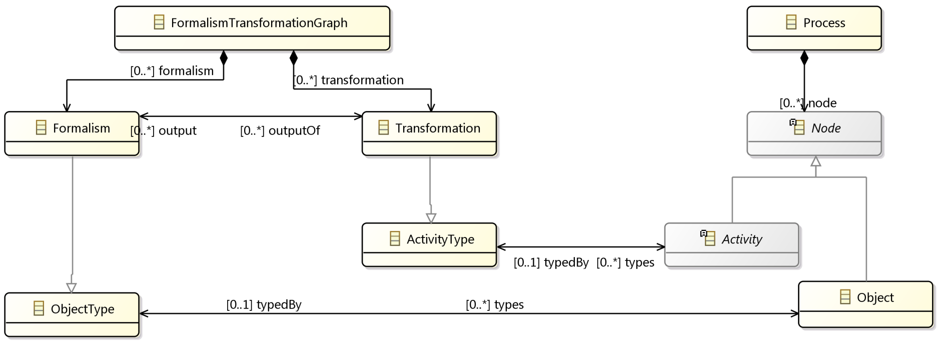


Figure 5.4: Excerpt of the FTG+PM part of the modeling formalism.

5.1.1.5 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 [60]. 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.

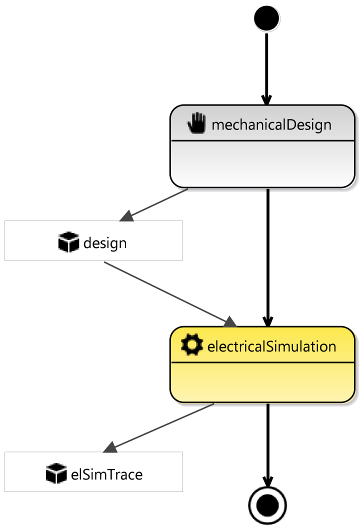


Figure 5.5: The PM of the running example.

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. [207]. The FTG for the process is shown in Figure 5.6.

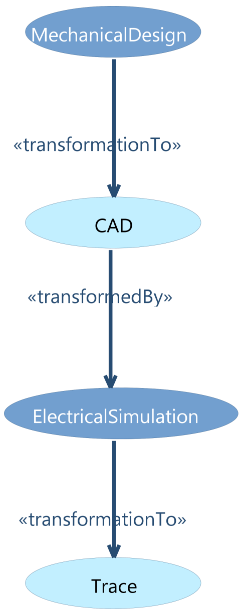


Figure 5.6: The FTG of the running example.

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.

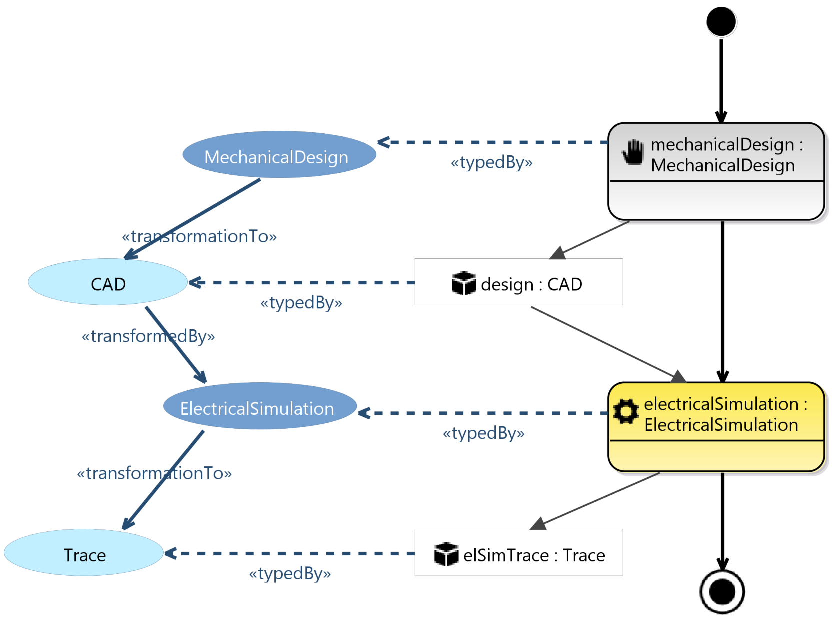


Figure 5.7: The FTG+PM of the running example.

5.1.2 Attributes, properties, constraints

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

By a property model we mean a tuple $\langle \Theta, P, R \rangle$ consisting of

- the set of attributes Θ; and
- the set of properties P; 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 [212].

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 [212]. In our approach, attributes and properties are treated uniformly.

We refer to the set of attributes and properties as *system characteristics*: $\Xi \equiv \Theta \bigcup P$.

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 running example, 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 the other way around.

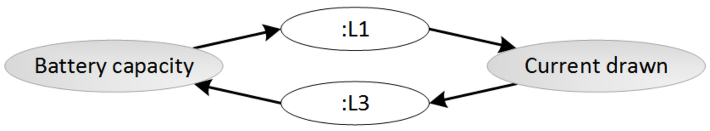


Figure 5.8: Influence relationship between two attributes.

A relationship r is formally defined as $r \in R = \langle \Xi_k^l, \Xi_m^n, \lambda \rangle$, i.e.

- a set of influencer (input) properties $\Xi_i^j = \{\xi_k..\xi_l\} \subseteq \Xi$,
- a set of influencee (output) properties $\Xi_m^n = \{\xi_m..\xi_n\} \subseteq \Xi$,
- 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 [46] and are as follows.

- $\mathbb{L}1$: the **fact of influence** is known, its extent is not;
- $\mathbb{L}2$: **sensitivity information** between two values is known;
- $\mathbb{L}3$: 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 $\mathbb{L}3$ influence on the *Battery capacity* as follows:

$$BatteryCapacity \geq \int CurrentDrawn(t)dt.$$

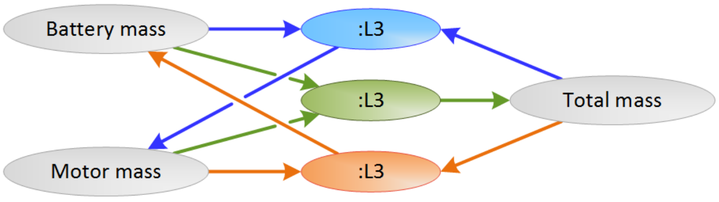
The relationship in the other direction, however, cannot be determined in such details and thus, only constitutes an $\mathbb{L}1$ relationship. In the running example, the relationship between the *Battery mass* and *Battery capacity* constitutes an $\mathbb{L}2$ relationship: increasing the capacity requires increasing the battery mass, although the exact relation cannot be provided as batteries come in various architectures.

Acausal influence relationships

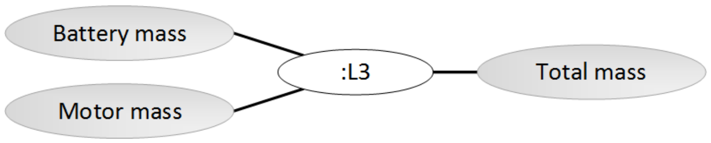
Acausality provides compactness in terms of the notation of relationships: it enables modeling of N-ary relationships in a more convenient and readable fashion. Figure 5.9a shows an N-ary influence relationship, depicting a simple law of physics:

$$BatteryMass + MotorMass = TotalMass.$$

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



(a) Causal notation of an N-ary relationship.



(b) Acausal notation of the same N-ary relationship.

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

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 ξ we mean a subset of system characteristic $\chi(\xi) \subseteq \Xi$ potentially affected by a change in ξ. Given two system characteristics ξ_i and ξ_j directly linked by a relationship $r = \langle \Xi_k^l, \Xi_m^n, \lambda \rangle$, the change set is defined as follows.

$$\xi_j \in \chi(\xi_i) \Leftrightarrow (\xi_i \in \Xi_k^l \wedge \xi_j \in \Xi_m^n)$$

That is, system characteristic ξ_j is in the change scope of system characteristic ξ_i iff ξ_i is an input system characteristic and ξ_j is an output system characteristic of a relationship. In the running example, the *Battery mass* system characteristic is in the change scope of the *Battery capacity* system characteristic.

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

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

$$\forall \xi_i, \xi_j, \xi_k \in \Xi : \xi_j \in \chi(\xi_i) \wedge \xi_k \in \chi(\xi_j) \Rightarrow \xi_k \in \chi(\xi_i),$$

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

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 as a tuple $\langle a, s, t_I \rangle$, where

- $a \in A$ is an activity;
- $s \in \Xi \bigcup R$ 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:

$$\forall s \in \Xi \bigcup R : s \in D,$$
$$\forall i \in I : i \in \Delta_d.$$

That is, attributes, properties and relationships are typed by appropriate languages, e.g. *OWL* languages [17], for modeling properties, or *graphs*, *algebra* and *Forrester system dynamics* [72] for modeling relationships.

Intents are typed by an *intent language* $T_I \in \mathcal{F}$. 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 shown an excerpt of it. The full, detailed metamodel, however, is shown in Figure A.3 of Appendix A.

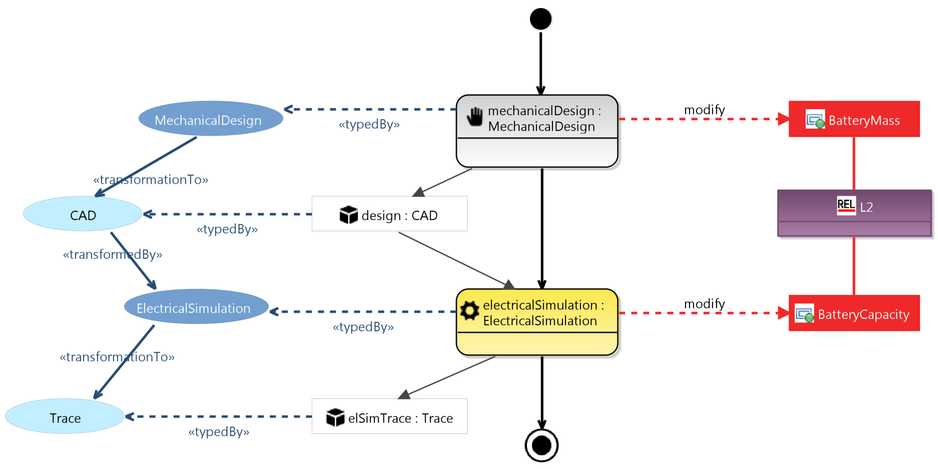


Figure 5.10: The running example in Figure 5.7, extended with attributes, constraints and intents.

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 *lectrical-Simulation* 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 in terms of the transit time. 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 $\Omega = \{\omega_1, \omega_2, \ldots\}$.

Availabilities. The availability b of resource defines how many units are available for the certain resource. $b(\omega) : \omega \in \Omega \to \mathbb{N}$. As defined by [11], if $b(\omega) = 1$, then ω 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 $B = \{b(\omega_1), b(\omega_2), \ldots\}$, i.e. a vector of natural numbers $B(\Omega) : \Omega \to \mathbb{N}^{|\Omega|}$.

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: $k(a) : a \in A \to \mathbb{N}^{|\Omega|}$, where the i-th element k_i is defined as $k_i(a, \omega) : a \in A, \omega \in \Omega \to \mathbb{N}$.

Allocation. The resource allocation l of an activity $a \in A$ and resource $\omega \in \Omega$ defines how many units of ω is a using, and is represented as a vector of natural numbers over the resources: $l(a) : a \in A \to \mathbb{N}^{|\Omega|}$, where the i-th element l_i is defined as $l_i(a, \omega) : a \in A, \omega \in \Omega \to \mathbb{N}$.

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. *Resource*s provide an integer attribute to capture the available units of that resource. (Note, that resources are unary on this metalevel, and the integer attribute on the instance level provides the cumulative semantics.) 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.

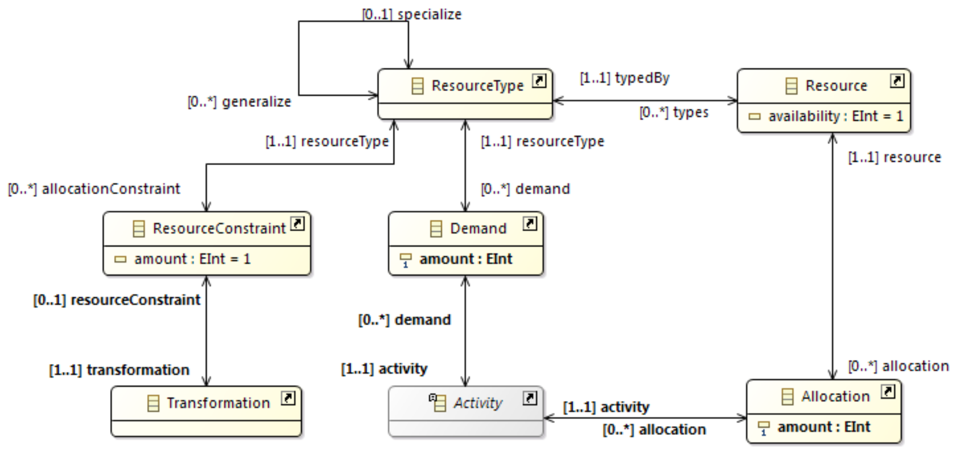


Figure 5.11: Excerpt of the resource modeling part of the formalism.

The demand is captured one meta-level above, as demands are stated against a specific type of resources. Hence the need for the explicit *ResouceType*. (Apart from the obvious need for a sound type system.) The *Demand* type then specifies the amount of units of a *ResouceType* 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 *ResourceConstraint*s an additional meta-level higher: on the level of *Transformation*s. 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.

Alternatively, DEVS could be used for handling resources and resource constraints, as discussed in Section 8.2.

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.

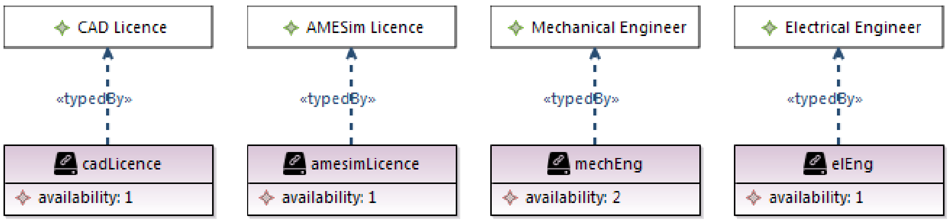
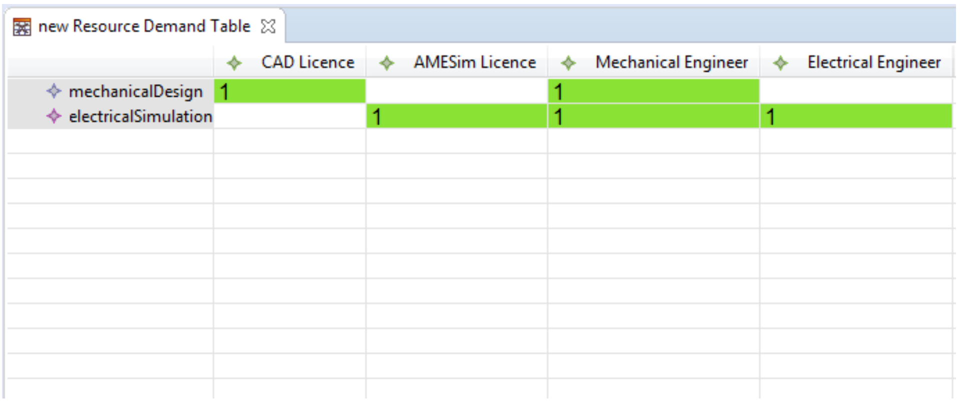


Figure 5.12: Resource types and resources of the running example.

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.

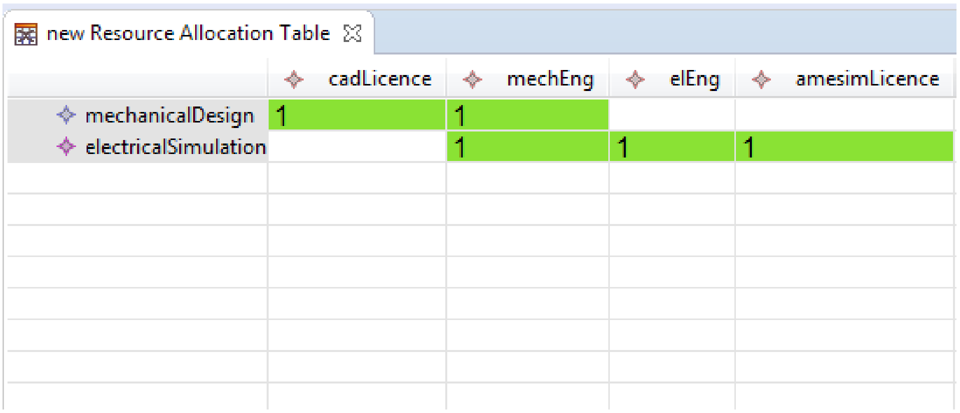


new Resource Demand Table

	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.


new Resource Allocation Table

	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 π, then is calculated as $C(\pi) = c_e(A) + c_u(A) + c_p$. Additionally, $C(\pi) : \pi \to \mathbb{R}^+$.

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.

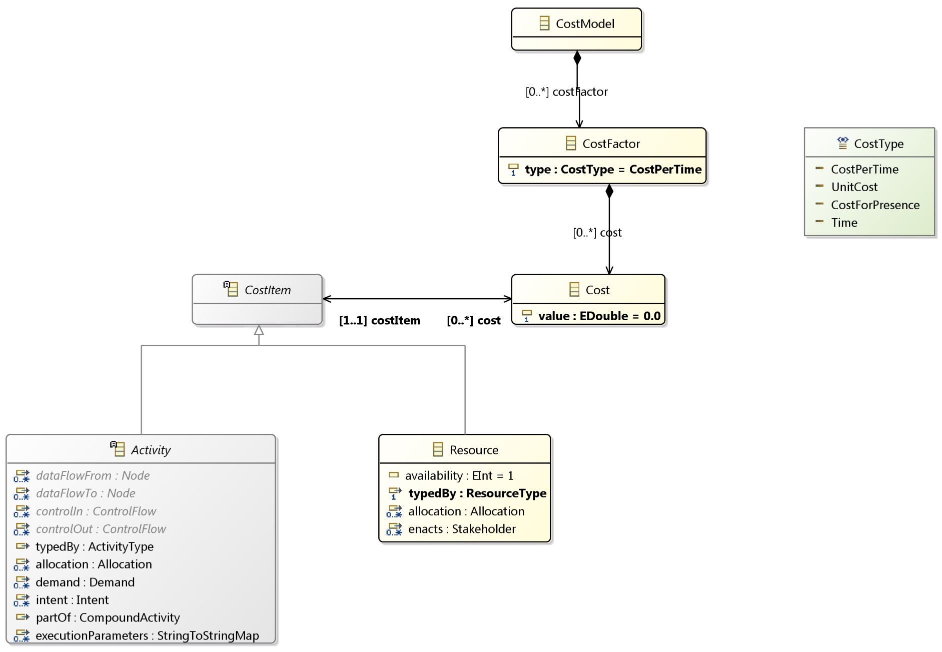
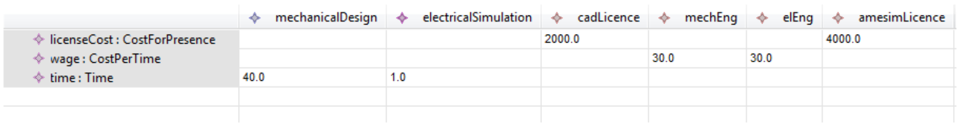


Figure 5.15: Excerpt of the cost modeling part of the formalism.

The formalism enables specifying various *CostFactor*s of different *CostType*s for the *CostItems*, i.e. *Activities* and *Resouces*.

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.



	mechanicalDesign	electricalSimulation	cadLicence	mechEng	elEng	amesimLicence
licenseCost : CostForPresence			2000.0			4000.0
wage : CostPerTime				30.0	30.0	
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: 40×30 in wages and 1×2000 in licence fees.
- The cost of the electrical design activity: $2 \times 1 \times 30$ in wages and 1×4000 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 work, 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 work: a formalism for modeling complex engineering processes. The majority of this section has been published in [45].

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 [164].

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 over the process π as follows.

$$\begin{aligned} \underset{\pi}{\text{minimize}} \quad & \mathrm{C}(\pi) \\ \text{subject to} \quad & \Psi \equiv \emptyset, \\ & v(\pi) = 1. \end{aligned}$$

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

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: $\forall a_1, a_2 \in A : par(a_1, a_2) \Leftrightarrow a_2 \notin \Delta_c^+(a_1) \wedge a_1 \notin \Delta_c^+(a_2)$.

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 $a_1, a_2 \in A : seq(a_1, a_2)$, system characteristic ξ is said to be exposed to a potential inconsistency due to insufficient inconsistency management in the following case.

$$\begin{gathered} \exists (\xi' \in \Xi, i_1, i_2 \in I) : i_1(a_1, \xi, t_I) \wedge i_2(a_2, \xi', t'_I) \wedge \\ \xi \in \chi^+(\xi') \wedge t_I = read \wedge t'_I = modify \Rightarrow \exists\, \psi(\xi) \in \Psi, \end{gathered}$$

where Ψ denotes the set of unmanaged inconsistencies, and $\psi(\xi)$ is the inconsistency mapping function of the system characteristic. $\Psi \mapsto \Psi \bigcup \{\xi\}$ iff there exists and inconsistency over ξ.

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

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

5.2.1.2 Parallel case

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

$$\exists(\xi' \in \Xi, i_1, i_2 \in I) : i_1(a_1, \xi, t_I) \wedge i_2(a_2, \xi', t'_I) \wedge \\ \xi \in \chi^+(\xi') \wedge t'_I = modify \Rightarrow \exists\, \psi(\xi) \in \Psi.$$

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 system characteristic 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* system characteristic ξ, while the other one *modifies* ξ'.

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. $a_1, a_2 \in A : seq(a_1, a_2) \Rightarrow \psi(\xi) \in \Psi$, 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. $a_1, a_2 \in A : par(a_1, a_2) \Rightarrow \psi(\xi) \in \Psi$, 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 [212], the stakeholders would agree on acceptance criteria of specific system characteristic *before* executing specific design activities. A special activity $a_{NegotiateContract}$ is added to the process to represent the contract negotiation phase. The activity accesses the unmanaged system characteristics with a *contract* intent. The contract is represented as an artifact. The contract is enforced during the affected part of the process, thus providing means to *avoid* inconsistencies.

In the current work, we assume both the contract negotiation activity, and the contract artifact as black boxes. The work of Vanherpen [211] provides appropriate means for including formalized contracts into the current approach.

5.2.2.4 Assumptions

A less rigorous approach to contracts is also possible by making an educated guess about the shared system characteristics. In the parallel case: $a_1, a_2 \in A : par(a_1, a_2) \Rightarrow \psi(\xi) \in \Psi$, one of the parallel activities makes assumptions about the system characteristics 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 Process optimization 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 π. 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 $\pi*$.

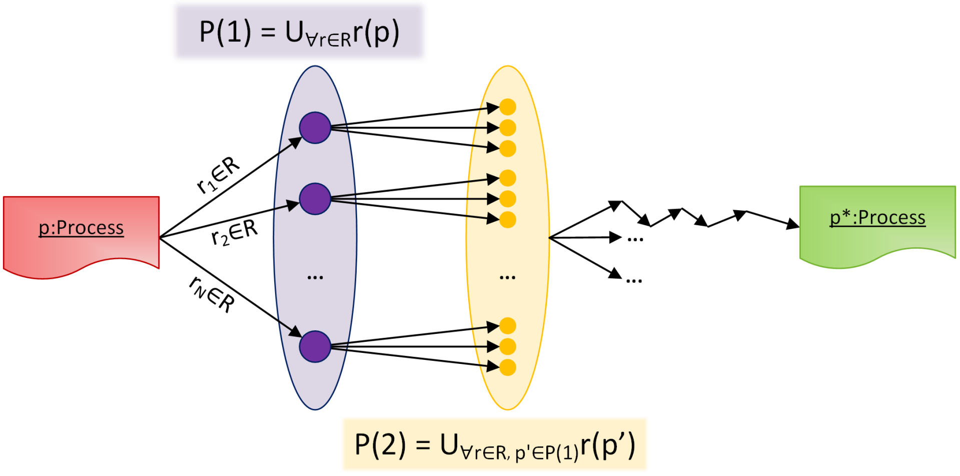


Figure 5.17: Detailed overview of the search process.

In step M, the set of process candidates is $\Pi(m) = \bigcup_{\forall r \in R, \pi' \in \Pi(m-1)} r(\pi')$. 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 Σ_Π, we mean the set of process candidates that are reachable from the initial process π via a sequence of transformation rules $r_1, \ldots r_m$. Formally, $\Sigma_\Pi = \bigcup_{i=1}^{m} \Pi(m) = \bigcup_{i=1}^{m} \bigcup_{\forall r \in R, \pi' \in \Pi(m-1)} r(\pi')$.

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 [86], or to obtain deeper insight into the design problem and better formulate the optimization problem [216]. 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 [166]. 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.

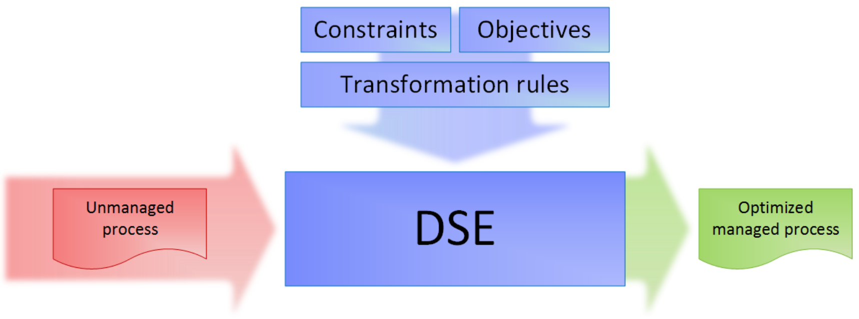


Figure 5.18: Structural overview of the DSE approach.

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.

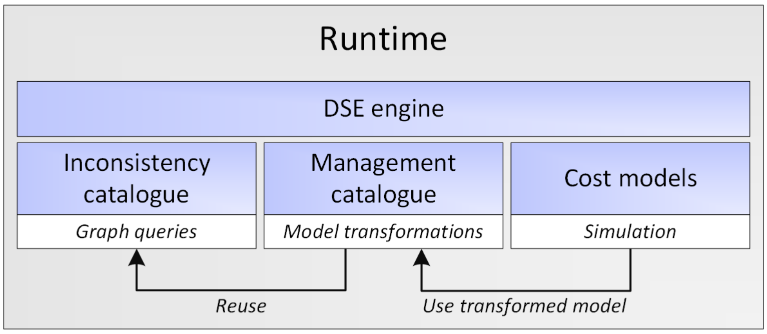


Figure 5.19: Architectural overview of the design space exploration component in the core of the prototype tooling.

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) [191] 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 4, system characteristics $p1 \in R^{+}(p2)$; and the activities $a2 \in \Delta_c(a1)$ accessing the two system characteristics with the respective *read* and *modify* intents are identified. Subsequently, in Lines 5-6, the number of applied inconsistency management patterns is determined. In case there is no management pattern applied (Line 7), 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 [20], that enable directly reusing the previously defined graph queries as the LHS.

5.2.3.3 Constraints and objectives

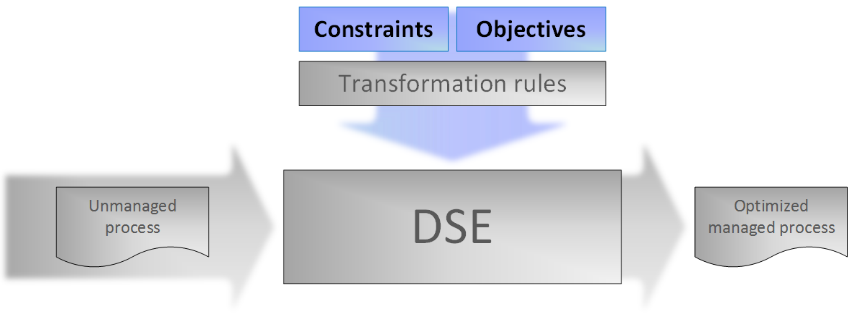


Figure 5.20: Constraints and objectives of the DSE.

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

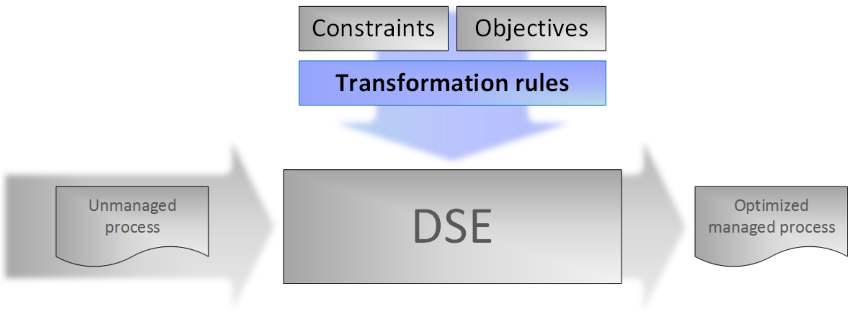


Figure 5.21: Transformation rules of the DSE.

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

$$\begin{gathered}\exists \xi \in \Xi : \psi(\xi) \in \Psi \wedge \\ \nexists m \in M : m(\psi(\xi)) \\ \rightarrow \mathit{apply}(m(\psi(\xi))), m \in M.\end{gathered}$$

That is, if an inconsistency pattern $\psi(\xi)$ is detected, 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.

```
rule manageWithContract(ic: unmanagedReadModify){
  // apply contract on ic
}
rule manageWithCheck(ic: unmanagedReadModify){
  // apply propertyCheck on ic
}
rule manageWithReorder(ic: unmanagedReadModify){
  // reorder(ic)
}
```

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

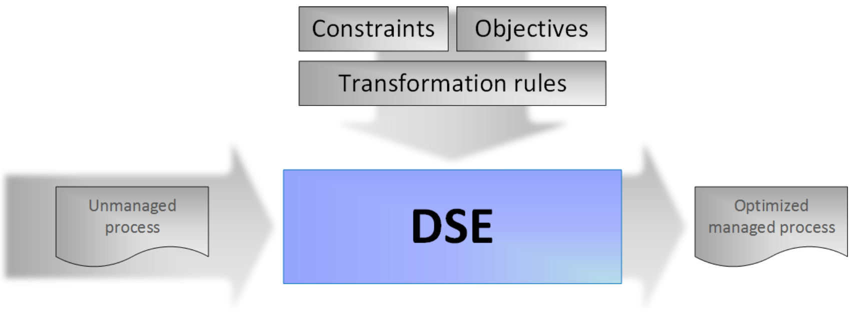


Figure 5.22: Design space exploration mechanism.

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 T'* out of every transformation rule *T* 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* $t \in T'$. 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 [86]. The search strategy takes the simulated performance value of the process as a guiding metric.

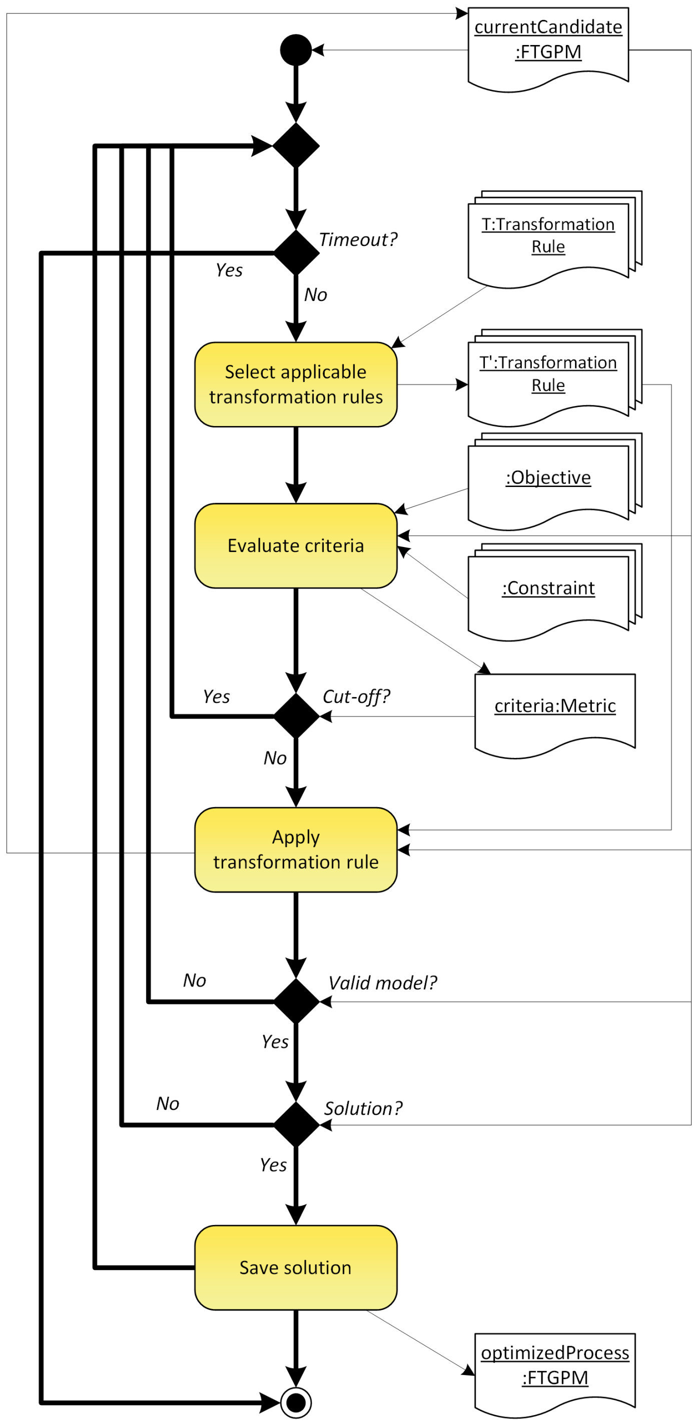


Figure 5.23: The PM of the DSE mechanism.

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

$$\forall \pi \in \Pi : C(\pi) = \sum_{1}^{i=|A(\pi)|} C(a_i)n(a_i).$$

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.

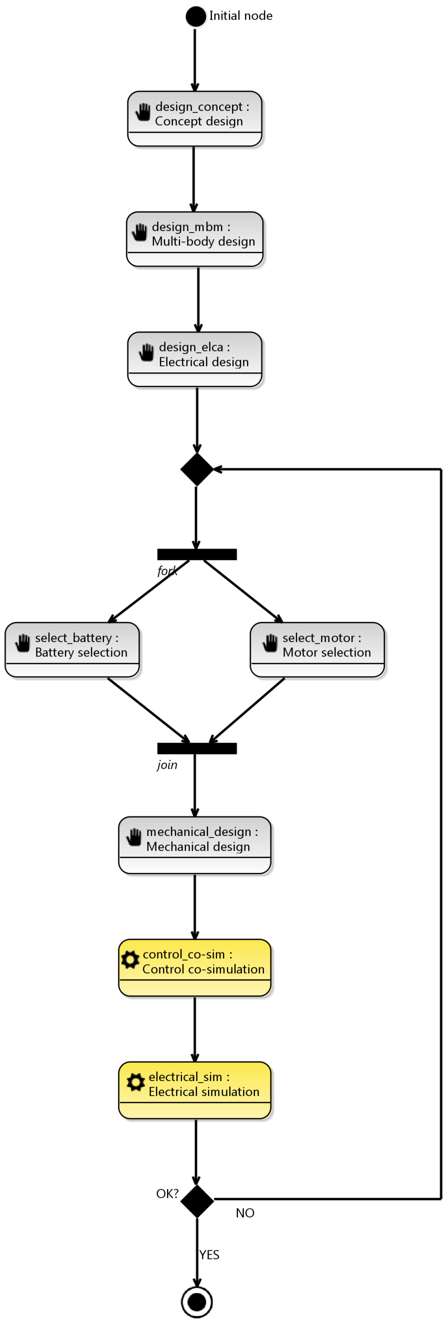
Algorithm 1 The fixed iteration simulation algorithm.

```
 1: procedure SIMULATE($N$ : int)                    ▷ $N$ is the number of iterations
 2:     $cost$ := 0
 3:     for $a1, a2 : A$ do
 4:         if loop($a1$, $a2$) then                  ▷ $a1$ and $a2$ form a loop
 5:             for $a3 : A \in$ loop($a1$, $a2$) do    ▷ $a3$ is in the loop
 6:                 $cost := cost + c(a3) \times N$    ▷ add cost of $a3$ to cost
 7:             end for
 8:         end if
 9:     end for
10:     return cost
11: end procedure
```

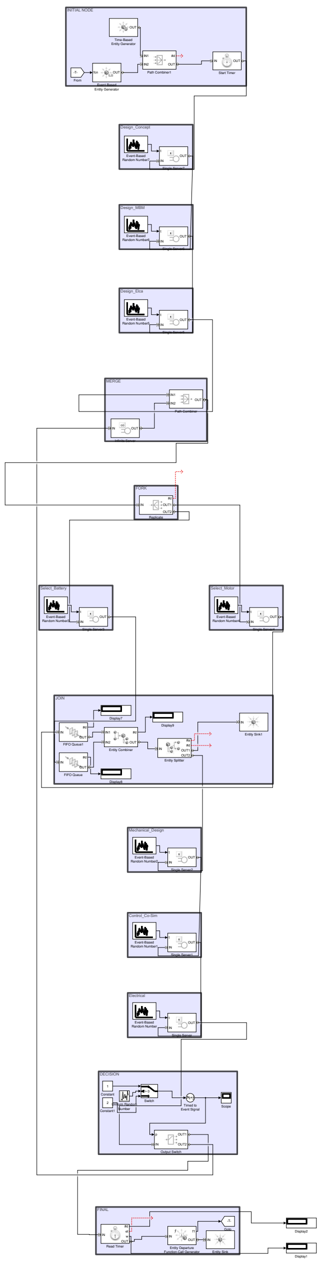
Event queueing network based simulation In event queueing network (EQN) based *stochastic simulations* [81], the decision of re-iterating over a loop is simulated with sampling from a probabilistic distribution. We carried out our early experiments using the SimEvents toolbox [127]. While stochastic 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 it 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.

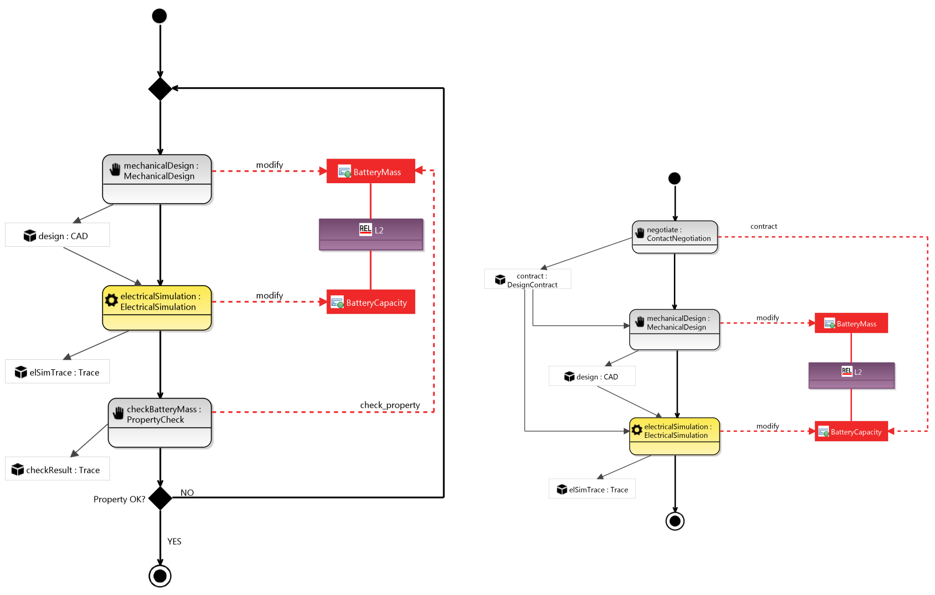

Initial node
design_concept : Concept design
design_mbm : Multi-body design
design_elca : Electrical design
fork
select_battery : Battery selection
select_motor : Motor selection
join
mechanical_design : Mechanical design
control_co-sim : Control co-simulation
electrical_sim : Electrical simulation
OK?
NO
YES

(a) Original process model.



(b) Quantitative SimEvents model.

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


mechanicalDesign : MechanicalDesign
modify
BatteryMass
L2
design : CAD
electricalSimulation : ElectricalSimulation
modify
BatteryCapacity
elSimTrace : Trace
checkBatteryMass : PropertyCheck
check_property
checkResult : Trace
Property OK?
NO
YES
negotiate : ContactNegotiation
contract
contract : DesignContract
mechanicalDesign : MechanicalDesign
modify
BatteryMass
L2
design : CAD
electricalSimulation : ElectricalSimulation
modify
BatteryCapacity
elSimTrace : Trace

(a) Property check (b) Contract

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 witch 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* [212]. 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 work: a formalism for modeling complex engineering processes. The majority of this section has been published in [47].

Process enactment is commonly defined as the use of software to support the execution of operational processes, which enables mixing automated and manual activities [37]. 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) [57] is used for modeling purposes, while the model transformations have been realized using the VIATRA framework [20].

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-

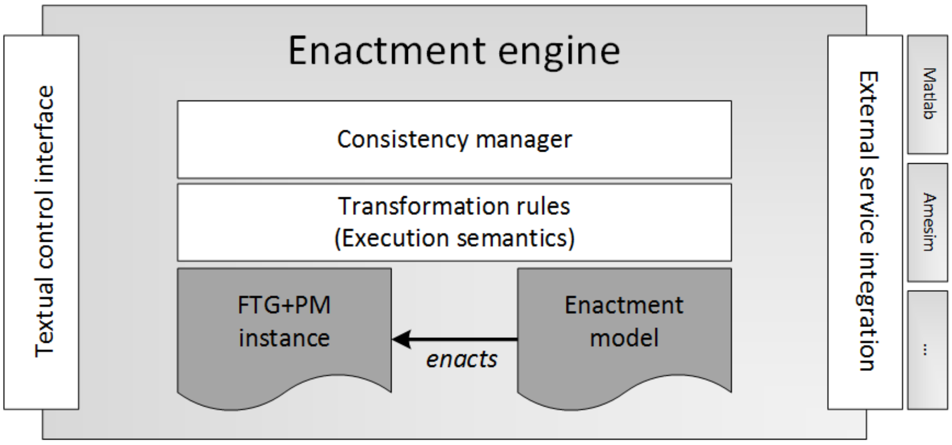


Figure 5.26: Architectural overview of the enactment engine.

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 [183] 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 [52]. 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 *Token*s define the marking of the process, i.e. the active *Node*s 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.

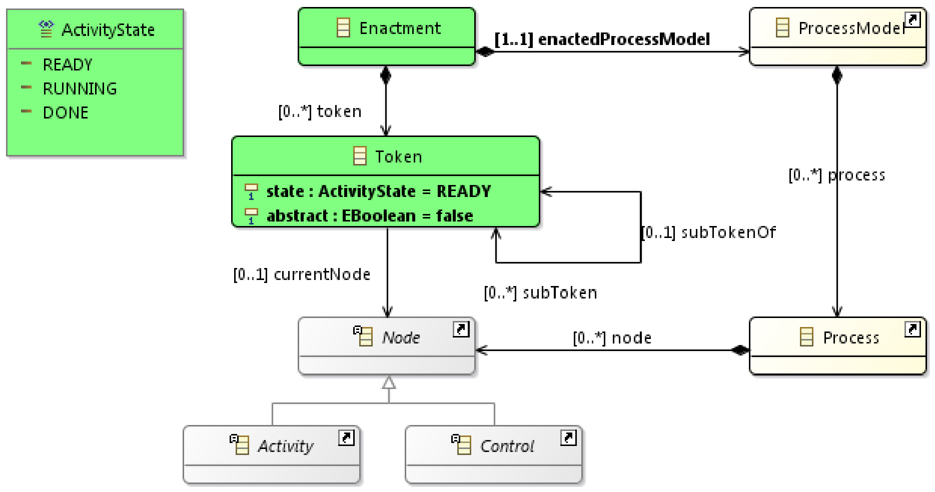


Figure 5.27: Metamodel for the enactment (green) along with the characteristic parts of the process metamodel.

- 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 μ of the process we mean the function $mu : N \to \mathbb{Z}$, where N denotes the set of the* Node*s of the process with an integer number $\mathbb{Z}$ of* Token*s 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.

```
pattern forkable(fork: Fork, token: Token){
  find tokenInNode(token, fork);
}

pattern tokenInNode(token : Token, node : Node){
  find activeToken(token);
  Token.currentNode(token, node);
}

pattern activeToken(token: Token){
  Token.abstract(token, a);
  a == false;
}
```

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 work: a formalism for modeling complex engineering processes. The majority of this section has been published in [47].

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 illustrative case of the AGV 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):

$$m_T = m_P + m_M + m_B. \tag{5.1}$$

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

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

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.

$$mass > 0\ [kg]. \tag{5.3}$$

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.

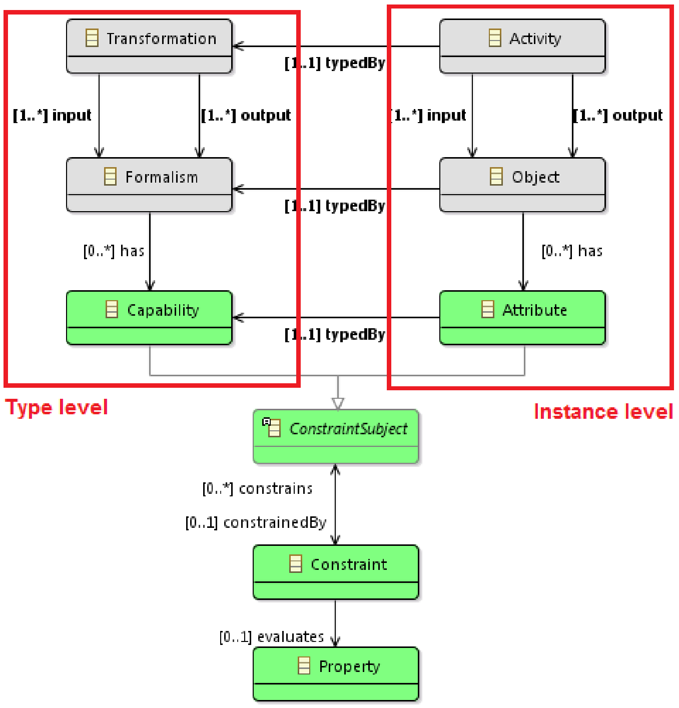


Figure 5.28: Excerpt from the extended FTG+PM metamodel supporting early online detection of inconsistencies.

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 γ of a formalism expresses the ability of a model, corresponding to the formalism, to reason about attributes.* $\forall\theta \in \Theta\exists\gamma \in \Gamma$: *typeof(θ)=γ.*

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 defined 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 $\mathbb{L}3$. 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 [60], 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.

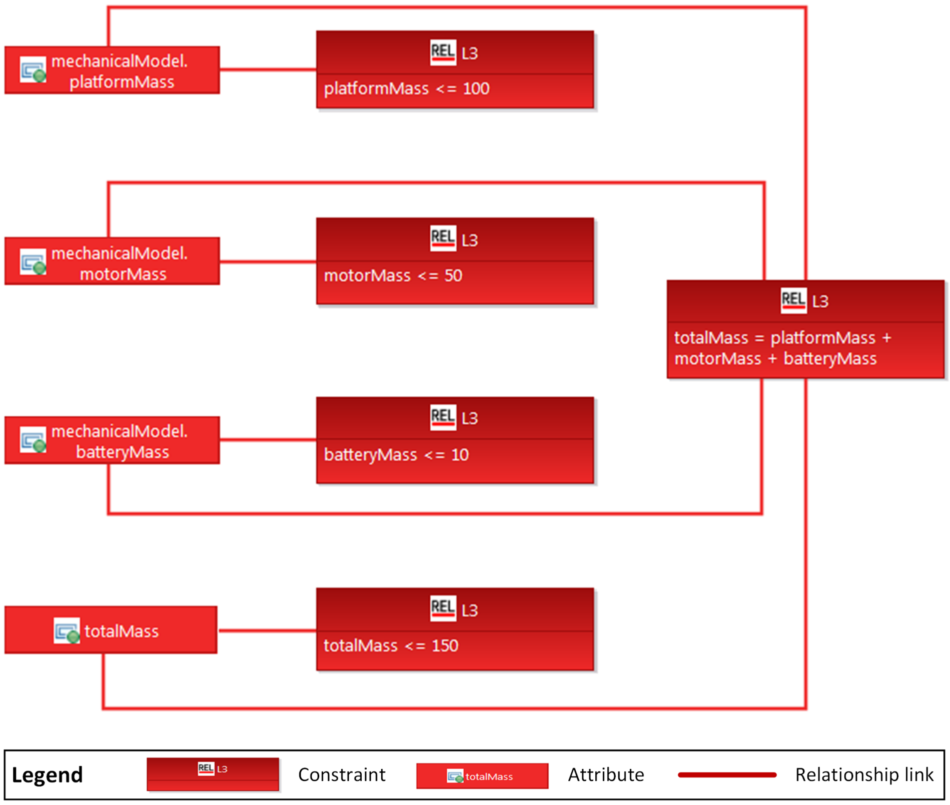


Figure 5.29: Attributes and constraints.

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 $\mathbb{L}3$. As defined in [46], the level of precision reflects what information a constraints carries:

- $\mathbb{L}1$: the **fact of influence** is known, its extent is not;
- $\mathbb{L}2$: **sensitivity information** between two values is expressed, e.g. by Forrester system dynamics [72];
- $\mathbb{L}3$: the constraint can be expressed using an exact **mathematical relationship**.

In this work, we assume $\mathbb{L}3$ 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.

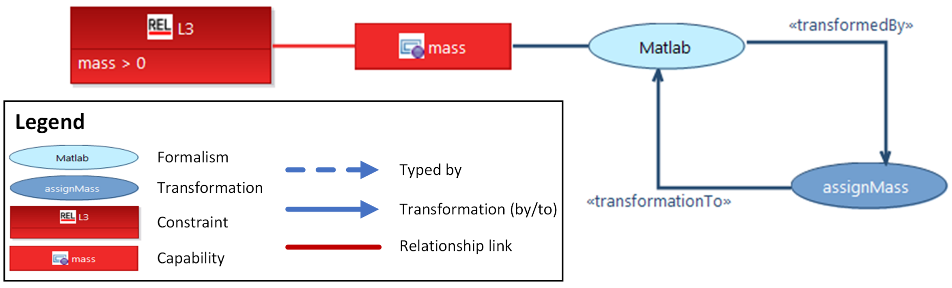


Figure 5.30: A capability and its constraint in the FTG.

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:

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

5.4.2.3 Properties

As presented in [212] 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.

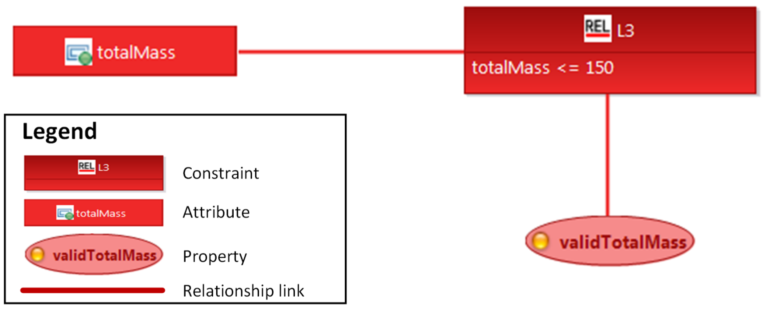


Figure 5.31: Excerpt from the example: the property *validMass*.

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 [212] in conjunction with our techniques, and using contract-based design [164] 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 w.r.t. *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 [45] 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.

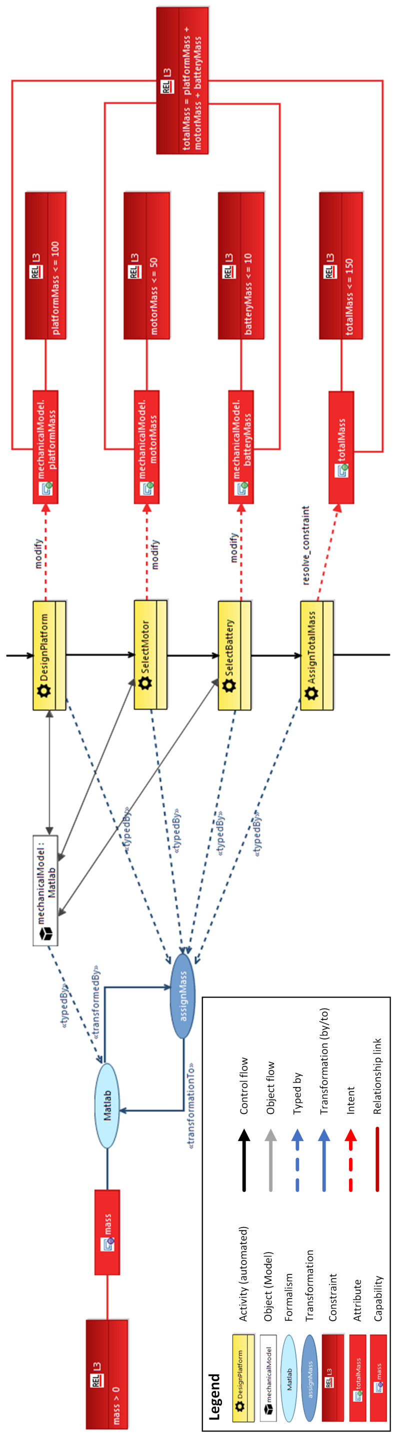


Figure 5.32: The FTG+PM based process model with the capabilities (left) and the attributes (right).

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:     $token.currentActivity \leftarrow nextActivity$            ▷ Move the token to the next activity
3:     for all $i$:Intent, $a$:Attribute: $i(token.currentActivity$, modify, $a$, $v)$ do
4:         UPDATEATTRIBUTEVALUE($a$, $v$)                          ▷ 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 [183] 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:                 $solution = solve(constraint1)$                ▷ 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], [66], [61], or [7].

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 $[kg] < m_P \leq$ 100 $[kg]$) 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 [37]. 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 work: a formalism for modeling complex engineering processes. The majority of this section has been published in [50].

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 [194]. 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 $d : W \times W \rightarrow \mathbb{R}$, [35], 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 [223]. Given its ability to support queueing systems and performance modeling [223], 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 $\langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ [206, 208], consisting of

- the set of input events X;
- the set of output events Y;
- the set of states $S : \times_{i=1}^{n} S_i$;
- the initial total state q_{init};
- the internal transition function $\delta_{int} : S \rightarrow S$;
- the external transition function $\delta_{ext} : S \rightarrow S$;
- the output function $\lambda : S \rightarrow Y \cup \{\phi\}$; and
- the time advancement function $ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$.

Coupled DEVS models are defined as a 7-tuple $\langle X_{self}, Y_{self}, D, MS, IS, ZS, select \rangle$, consisting of

- X_{self};
- Y_{self};
- the set of model instances D;
- the set of model specifications MS;
- the set of model influencees IS;
- the set of translation functions ZS; and
- the tie-breaking function $select : 2^{D} \rightarrow D$.

To keep our approach as general as possible, we do not tie ourselves to one particular process modeling formalim, but rather, we provide a mapping of each of the original 20 *van der Aalst* process/workflow patterns [194] 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 [207], 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.

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) [45]. 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 Simulation*s 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.

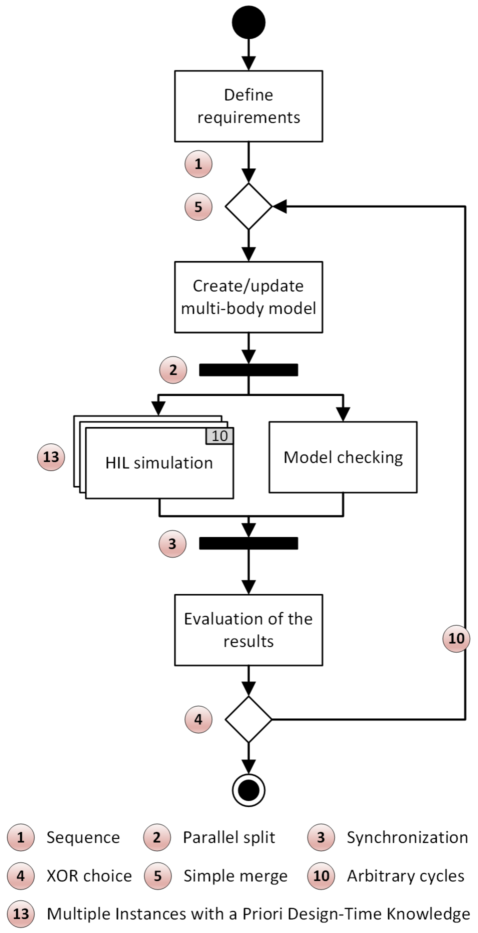


Figure 5.33: The process of selecting the optimal hardware component.

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 [122]. 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 handcoded 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 [122] 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.

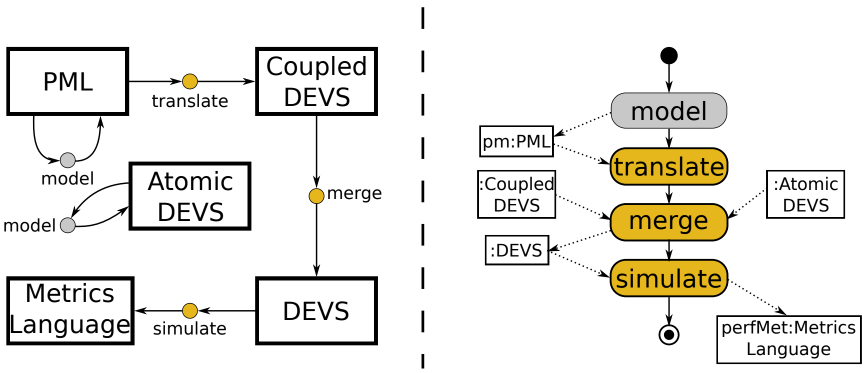


Figure 5.34: FTG+PM model of our approach.

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 [194]. 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).

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

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

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

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

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.

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

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

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

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

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).

5.5.1.3 Implementation

The approach was implemented in the Modelverse [207], 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

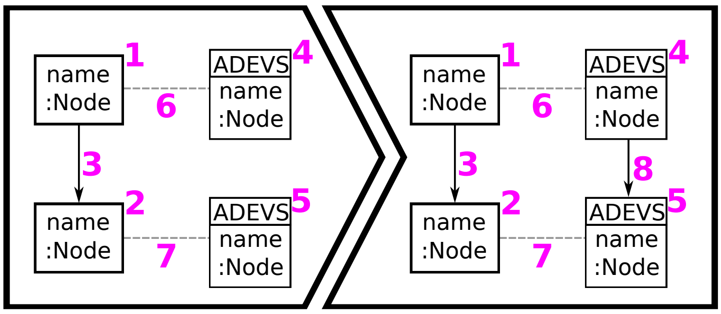


Figure 5.35: Transformation rule for control flow links.

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 [205], our DEVS simulator, through the use of external services [202]. 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 worklow, 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 also augmented with a quantitative *decision function*.

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 σ relative to mean μ, 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

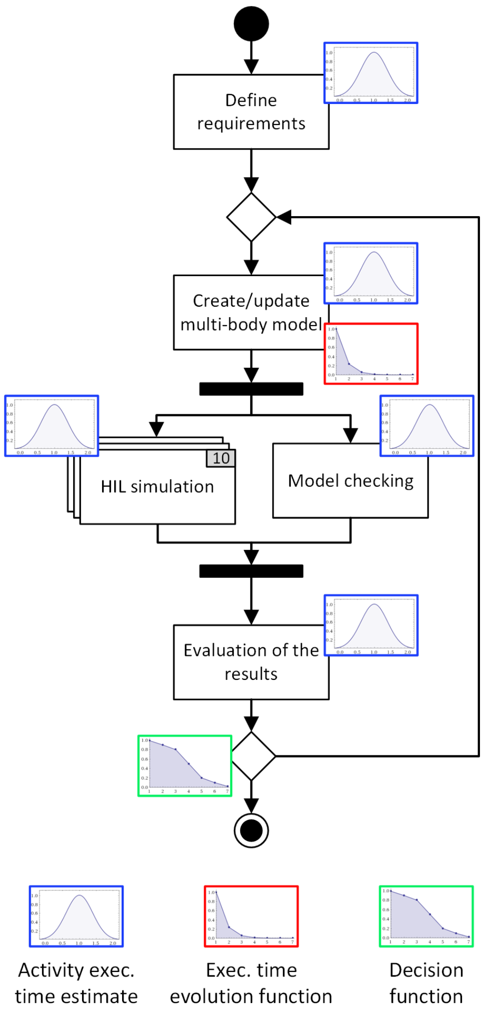

Define requirements
Create/update multi-body model
HIL simulation
10
Model checking
Evaluation of the results
Activity exec. time estimate
Exec. time evolution function
Decision function

Figure 5.36: The augmented process.

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 *resouce_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).

$$Activity = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{5.9}$$

$$\begin{aligned}
X &= \{ControlToken, Resources\} \\
Y &= \{ControlToken, Request, Release\} \\
S &= \{active, inactive, request, waiting\} \\
q_{init} &= (inactive, 0.0) \\
\delta_{int} &= \{active \rightarrow inactive, request \rightarrow waiting\} \\
\delta_{ext} &= \{((inactive, \_), ControlToken) \rightarrow request, \\
&\quad ((waiting, \_), Resources) \rightarrow active\} \\
\lambda &= \{active \rightarrow [ControlToken, Release], \\
&\quad request \rightarrow [Request]\} \\
ta &= \{active \rightarrow t_{process}, inactive \rightarrow \infty, \\
&\quad request \rightarrow 0, waiting \rightarrow \infty\}
\end{aligned}$$

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.

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).

$$ResourceHandler = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{5.10}$$

$$\begin{aligned}
X &= \{Request, Release\} \\
Y &= \{Resources\} \\
S &= [Request] \times \mathbb{N} \\
q_{init} &= (([], resources), 0.0) \\
\delta_{int} &= \{([R, r], i) \rightarrow ([R], i-1)\} \\
\delta_{ext} &= \{((([R], i), e), Request) \rightarrow ([R, Request], i), \\
&\quad ((([R], i), e), Release) \rightarrow ([R], i+1)\} \\
\lambda &= \{([R, r], i) \rightarrow Resources(r)\} \\
ta &= \{([R], i > 0) \rightarrow 0, \\
&\quad ([], i) \rightarrow \infty, \\
&\quad ([R], 0) \rightarrow \infty\}
\end{aligned}$$

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.

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,

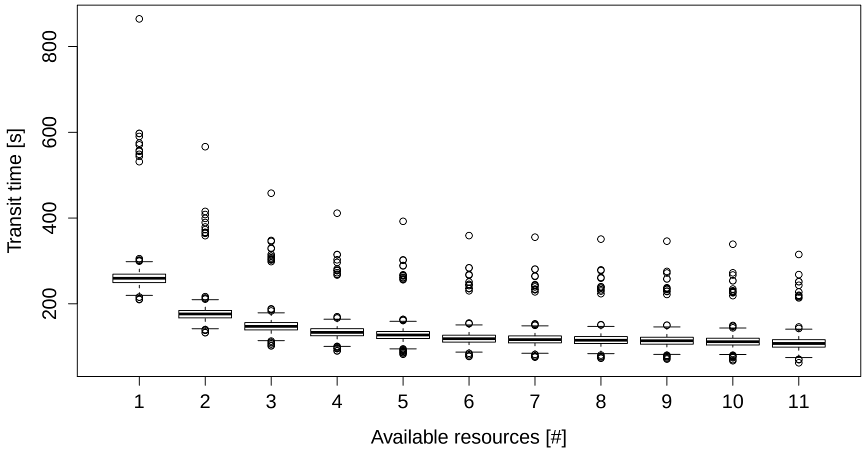


Figure 5.37: Results of the simulations.

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 [194]. 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 [45], where inconsistencies and their quantitative metrics [46] 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 work: a formalism for modeling complex engineering processes. The majority of this section has been published in [202]. The results of this section have not been integrated into the prototype tooling. As opposed to the rest of the contributions, a more simplistic illustrative example (Section 5.6.2) is used in order to keep the contributions understandable. (See Figure 5.39.)

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 [146]. 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 [203], a variant of Statecharts [84]. 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 [84]. 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 [203] 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 [223]. 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.

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 [207, 204], our prototype Multi-Paradigm modeling tool. Simulations were performed using PythonPDEVS [205] as an external service.

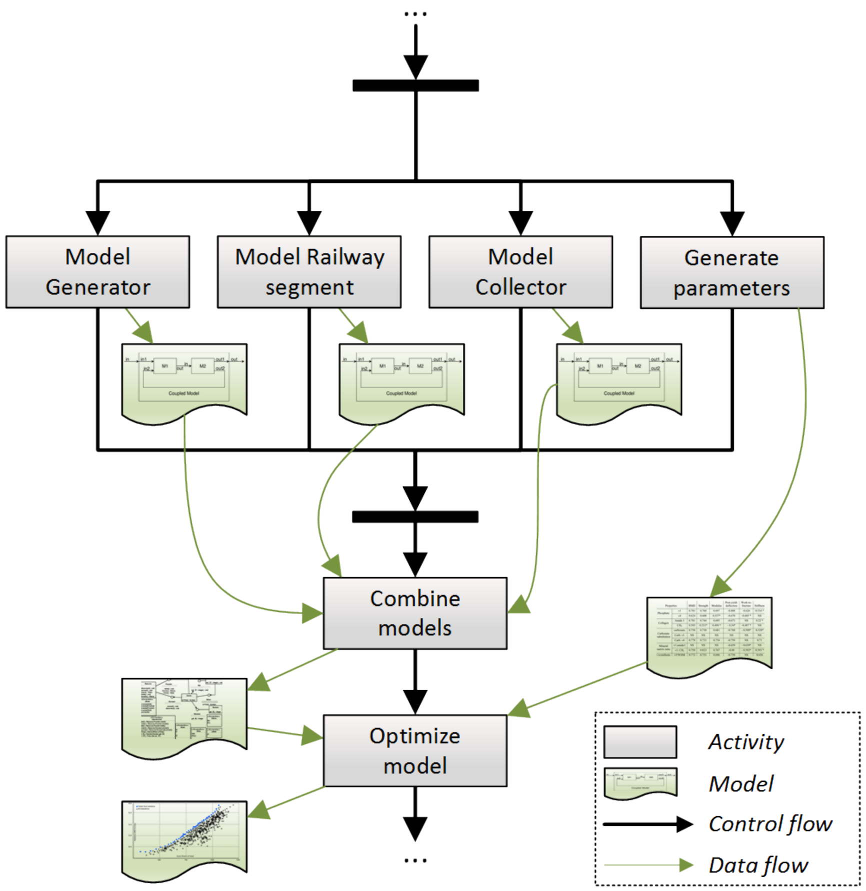


Figure 5.38: The process model of the example.

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.

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.

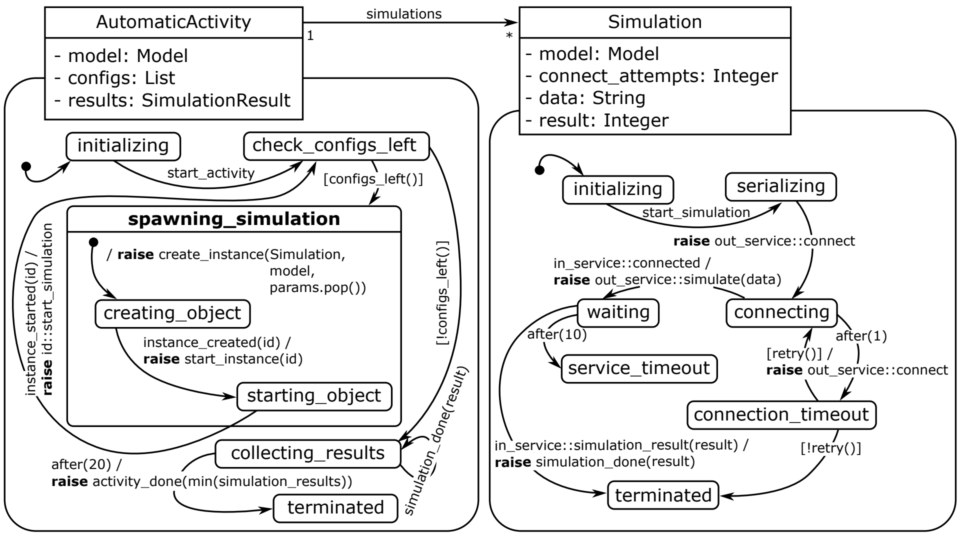

AutomaticActivity
- model: Model
- configs: List
- results: SimulationResult
simulations
1
*
Simulation
- model: Model
- connect_attempts: Integer
- data: String
- result: Integer
initializing
check_configs_left
start_activity
[configs_left()]
spawning_simulation
/ raise create_instance(Simulation, model, params.pop())
creating_object
instance_created(id) / raise start_instance(id)
starting_object
instance_started(id) / raise id::start_simulation
[!configs_left()]
collecting_results
simulation_done(result)
after(20) / raise activity_done(min(simulation_results))
terminated
initializing
serializing
start_simulation
raise out_service::connect
in_service::connected / raise out_service::simulate(data)
waiting
connecting
after(10)
after(1)
service_timeout
[retry()] / raise out_service::connect
connection_timeout
in_service::simulation_result(result) / raise simulation_done(result)
[!retry()]
terminated

Figure 5.39: Automated activity: protocol implemented to communicate with an external service.

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 [170]. 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.

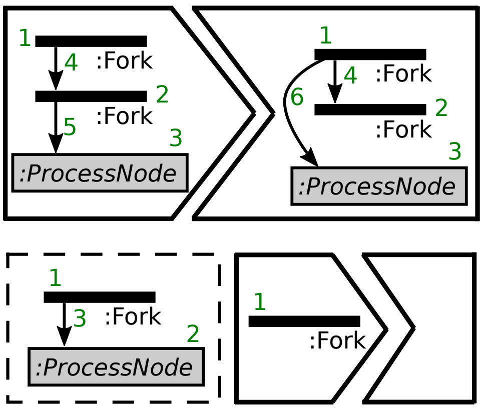


Figure 5.40: Optimize rules.

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.

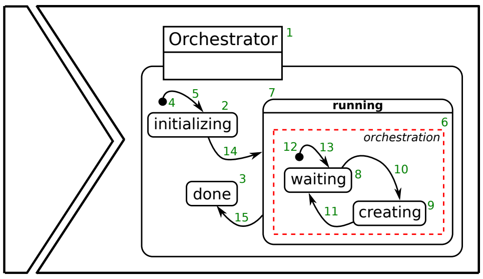


Figure 5.41: Orchestrator rule.

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).

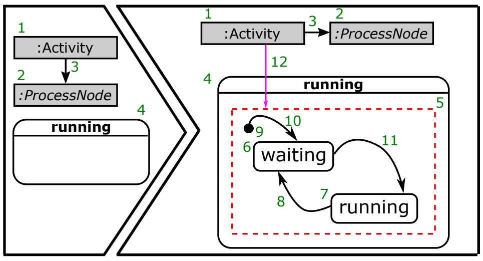


Figure 5.42: Activity rule.

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 it successors simultaneously.

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.

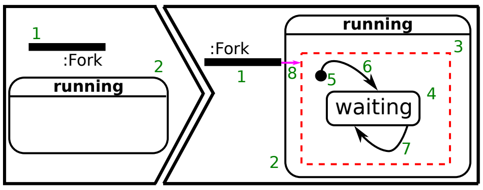


Figure 5.43: Fork rule.

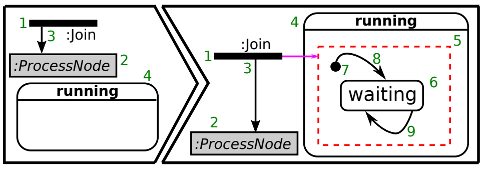


Figure 5.44: Join rule.

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.

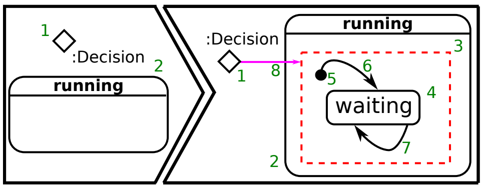


Figure 5.45: Decision rule.

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.

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

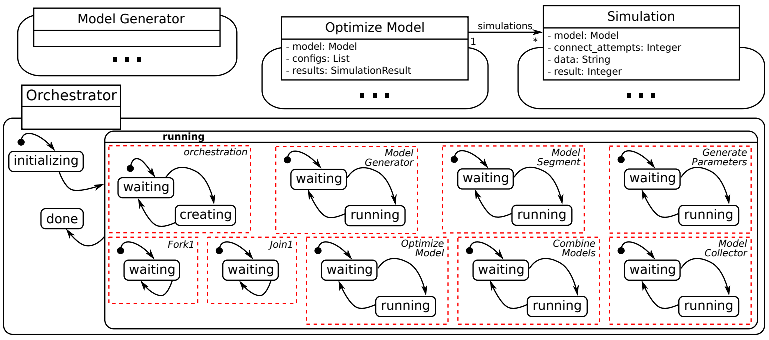


Figure 5.46: Process model from Figure 5.38 mapped to SCCD.

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 [150] or debugged [136], 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 [45] 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.

Chapter 6

Proof of concept

In this section, we elaborate on the utility of our methods through two demonstrative examples. Both examples are of a system engineering nature and are based on the engineering process of a real-life autonomous guided vehicle (AGV).

The two examples are discussed in Section 6.2 and Section 6.3, respectively.

6.1 Introduction

Figure 6.1 shows the automated guided vehicle, of which engineering process is referred in our demonstrative examples later in this chapter.

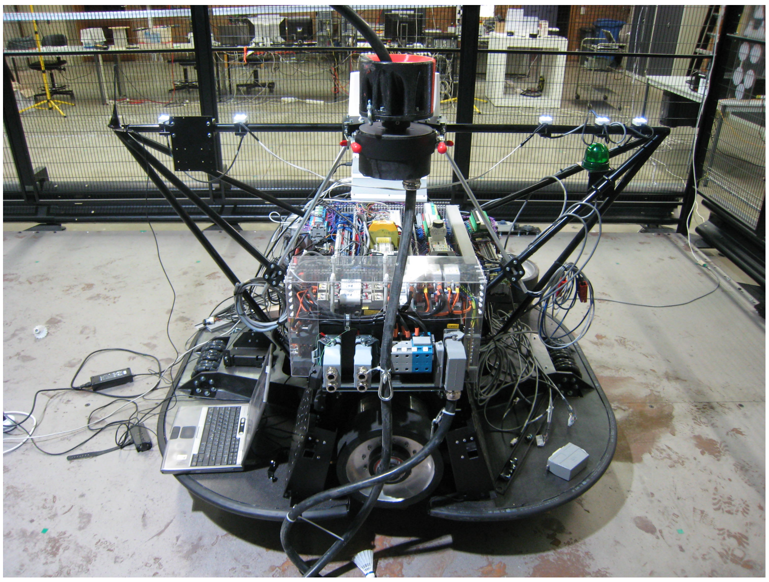


Figure 6.1: The automated guided vehicle (AGV) discussed in our demonstrative examples.

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.

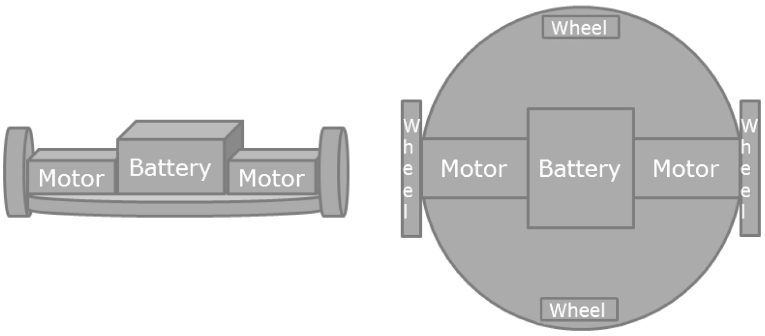


Figure 6.2: Front and top view of the conceptual design of the AGV.

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.

```matlab
%%Initial conditions, variables, etc
designSteps = containers.Map();
    % Motor selection design step
motorSelection.parameters = {'Pdes';'Peff';'m';'r';'l'};
motorSelection.inputs = {'Pdes'};
motorSelection.outputs = {'m';'l';'r';'Peff'};
motorSelection.analytical = motorAnalytical;
motorSelection.functionCall = '[Peff,m,l,r] = MotorSelectionF(Pdes)';
motorSelection.data = containers.Map();
designSteps('motorSelection') = motorSelection;
    % Battery selection design step
batterySelection.parameters = {'Cdes';'Ceff';'m';'l';'w';'h'};
batterySelection.inputs = {'Cdes'};
batterySelection.outputs = {'m';'l';'w';'h';'Ceff'};
batterySelection.analytical = batteryAnalytical;
batterySelection.functionCall = '[Ceff,m,l,w,h] = BatterySelectionF(Cdes)';
batterySelection.data = containers.Map();
designSteps('batterySelection') = batterySelection;
%%(...)

%%Iterations of the engineering process

    % ITERATION 1
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint;call(motorSelection,motorSelectionPoint)];
add(motorSelection.data,motorSelectionPoint);

batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint=[batterySelectionPoint;call(batterySelection,
    ↪ batterySelectionPoint)];
add(batterySelection.data,batterySelectionPoint);

assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
add(mechDesign.data,mechDesignPoint);
%%(...)

    % ITERATION 2
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint;call(motorSelection,motorSelectionPoint)];
add(motorSelection.data,motorSelectionPoint);

batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint=[batterySelectionPoint;call(batterySelection,
    ↪ batterySelectionPoint)];
add(batterySelection.data,batterySelectionPoint);
%%(...)

    % ITERATIONS 3-7
%%(...)

%%Function definitions
%%(...)
```

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 demonstrative example 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 demonstrative example, 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 demonstrative example. To reproduce the original Matlab-encoded workflow, only the PM and the typing FTG are needed.

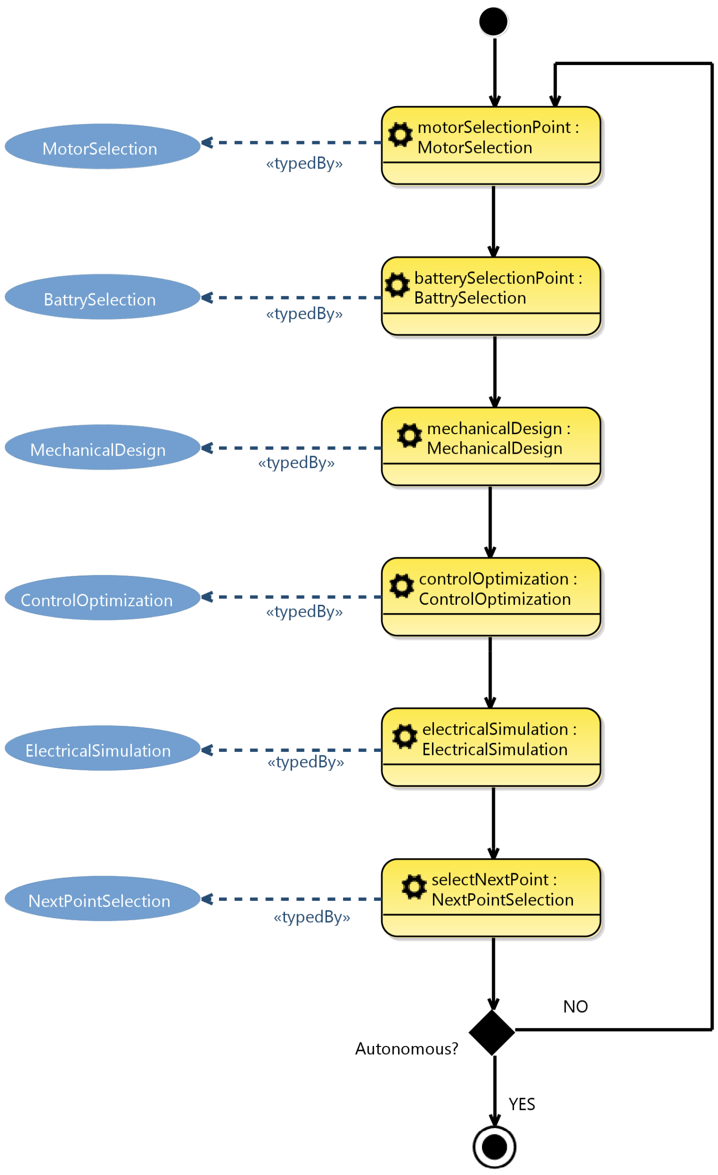
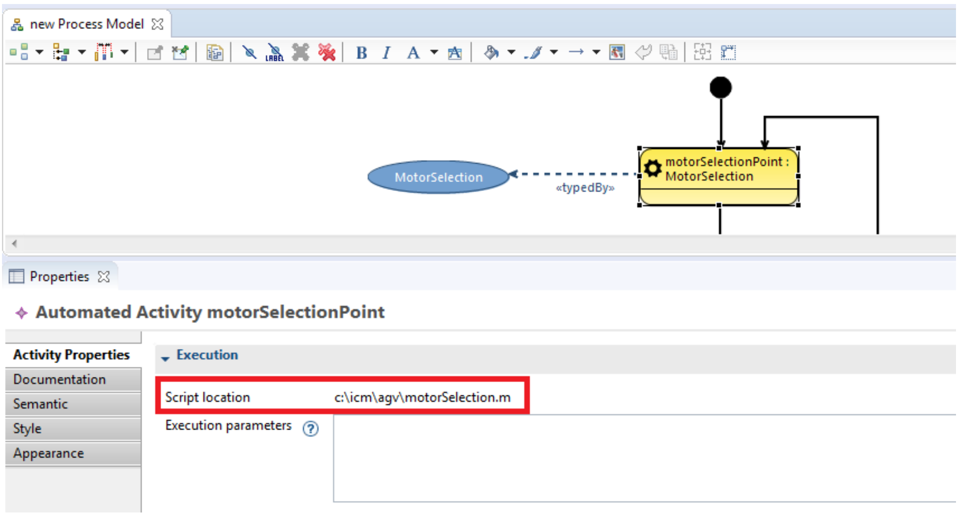

MotorSelection
«typedBy»
motorSelectionPoint : MotorSelection
BattrySelection
«typedBy»
batterySelectionPoint : BattrySelection
MechanicalDesign
«typedBy»
mechanicalDesign : MechanicalDesign
ControlOptimization
«typedBy»
controlOptimization : ControlOptimization
ElectricalSimulation
«typedBy»
electricalSimulation : ElectricalSimulation
NextPointSelection
«typedBy»
selectNextPoint : NextPointSelection
NO
Autonomous?
YES

Figure 6.3: The FTG+PM of the demonstrative example.

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.)



Script location c:\icm\agv\motorSelection.m

Execution parameters

Figure 6.4: Typed activity with the Matlab script *motorSelection.m* providing the execution semantics.

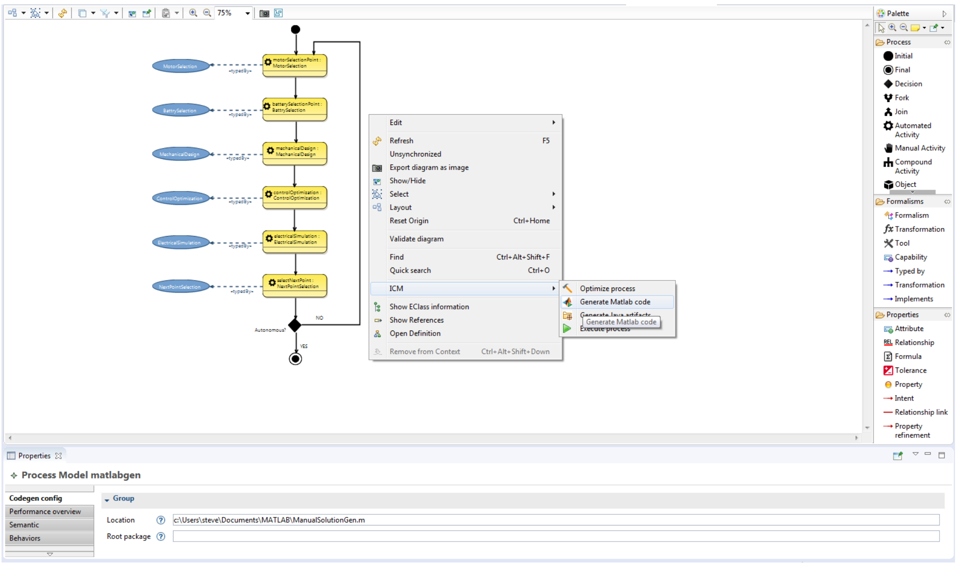
The `motorSelection.m` script is fairly simple, as shown in Listing 4.

Listing 4 Matlab script providing semantics for the *MotorSelection* transformation.

```
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
call(motorSelection,'motorSelectionPoint');
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.



Properties

Process Model matlabgen

Codegen config
Performance overview
Semantic
Behaviors

Group
Location: c:\Users\steve\Documents\MATLAB\ManualSolutionGen.m
Root package:

Figure 6.5: The built-in functionality for generating Matlab scripts shown in the menu.

6.3 Modeling by our approach

In our second demonstrative example, we take previously elaborated example 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. Informal discussions and collaborative modeling sessions have been conducted to access the tacit knowledge in form of socialization [138]. As suggested by Hutchinson et al. [94], such an approach works well and with a minimal bias in case of MDE professionals.

In this section, we aim at demonstrating 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 demonstrative example 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.

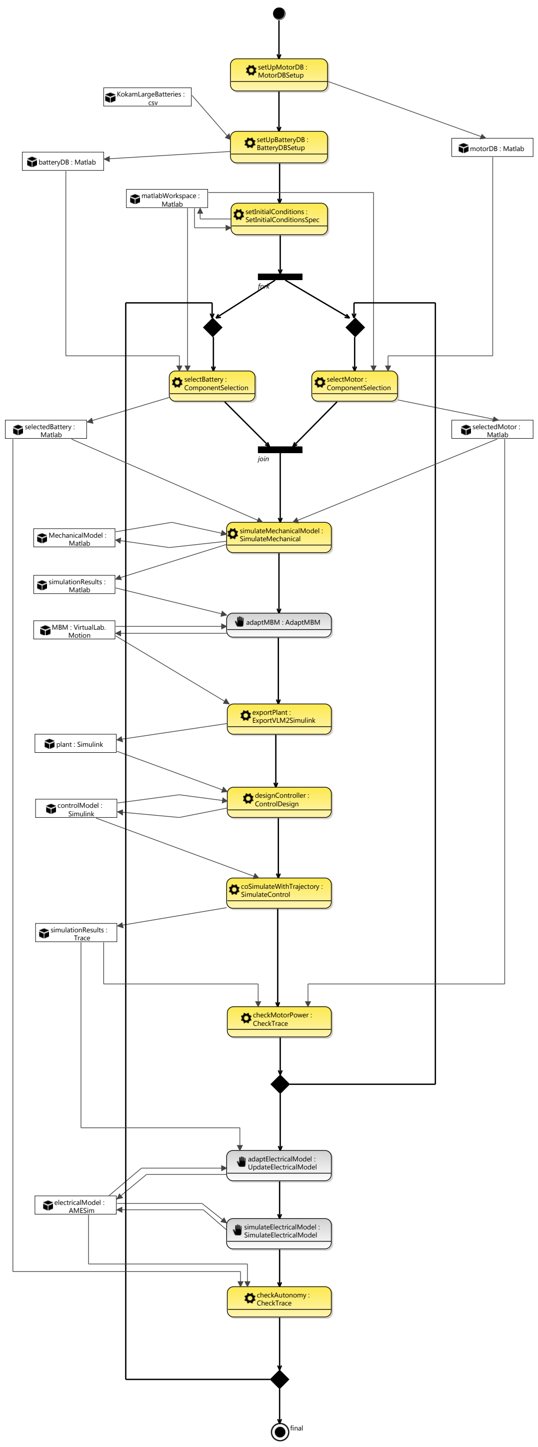

setUpMotorDB : MotorDBSetup
KokamLargeBatteries : csv
setUpBatteryDB : BatteryDBSetup
motorDB : Matlab
batteryDB : Matlab
matlabWorkspace : Matlab
setInitialConditions : SetInitialConditionsSpec
fork
selectBattery : ComponentSelection
selectMotor : ComponentSelection
selectedBattery : Matlab
selectedMotor : Matlab
join
MechanicalModel : Matlab
simulateMechanicalModel : SimulateMechanical
simulationResults : Matlab
adaptMBM : AdaptMBM
MBM : VirtualLab. Motion
exportPlant : ExportVLM2Simulink
plant : Simulink
designController : ControlDesign
controlModel : Simulink
coSimulateWithTrajectory : SimulateControl
simulationResults : Trace
checkMotorPower : CheckTrace
adaptElectricalModel : UpdateElectricalModel
electricalModel : AMESim
simulateElectricalModel : SimulateElectricalModel
checkAutonomy : CheckTrace
final

Figure 6.6: The PM of the demonstrative example.

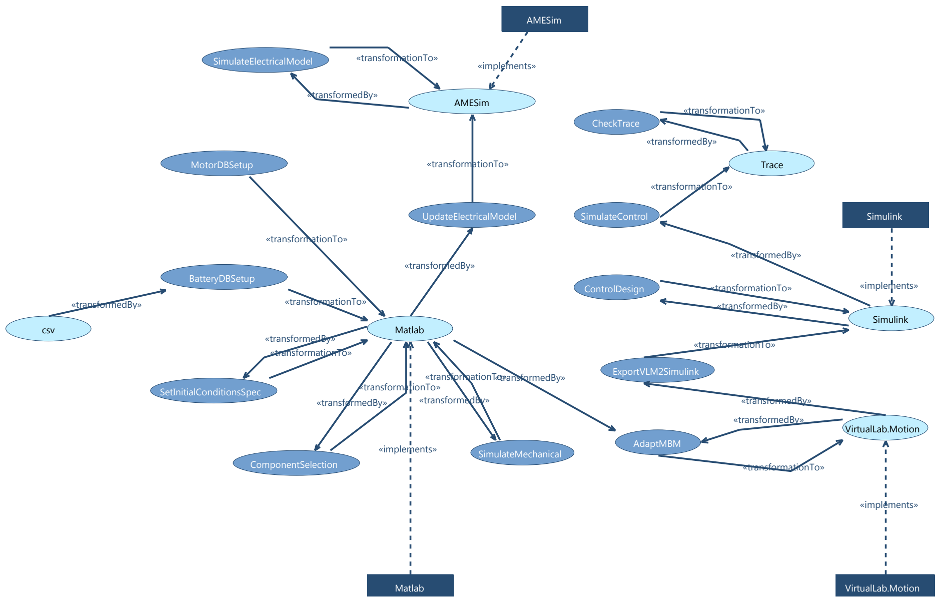


Figure 6.7: The FTG of the demonstrative example.

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.

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 [126].

The activity takes the previously obtained simulation results, and updates the multi-body model in the VirtualLab.Motion tool [186]. 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 the 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 [173].

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:

$$capacity \geq \int currentDrawn(t)dt.$$

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.

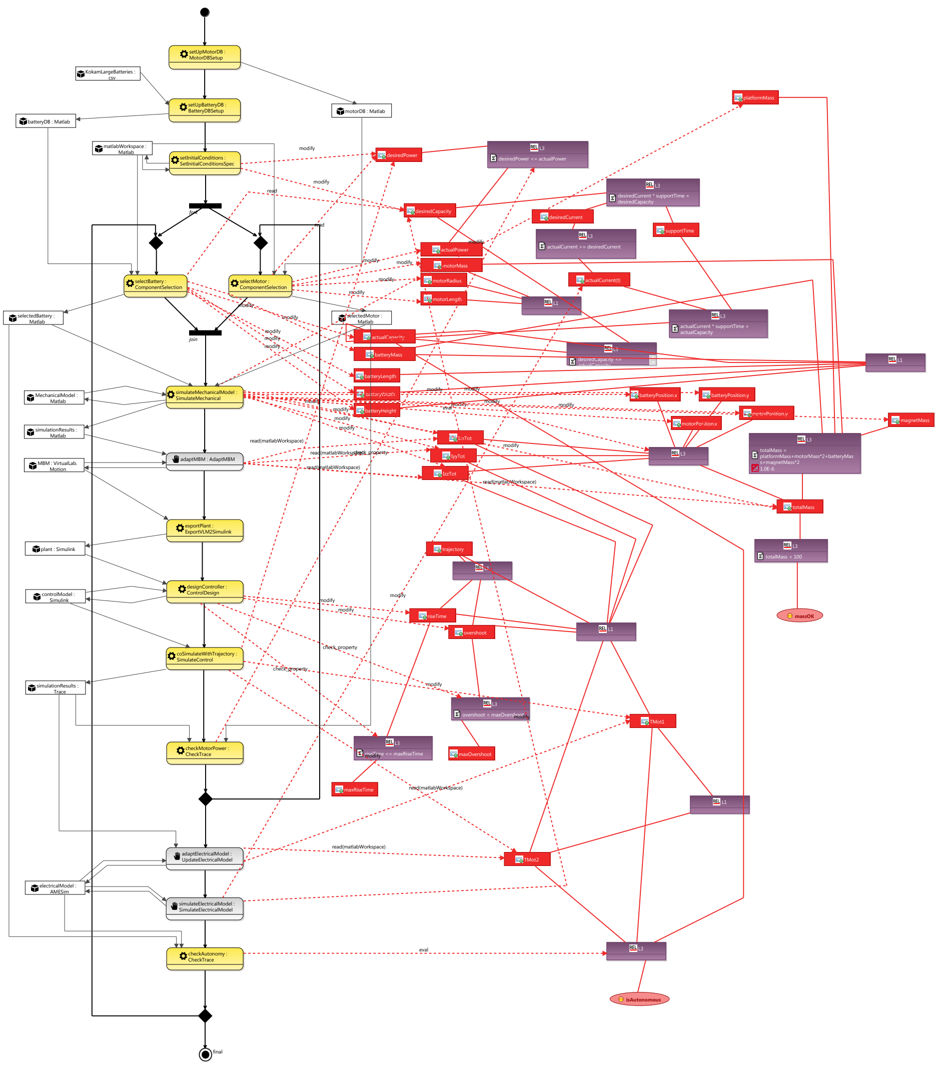


Figure 6.8: The augmented PM of the demonstrative example.

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 demonstrative example.

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:

- *desiredPower* $\leq$ *actualPower*
- *desiredCapacity* $\leq$ *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 *totalMass* = *platformMass* + *motorMass**2 + *batteryMass* + *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 [49].

6.3.3 Properties

For demonstrative reasons, we also provide two properties. Both the *massOK* and *isAutonomous* properties are derived from a performance value [212] 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.

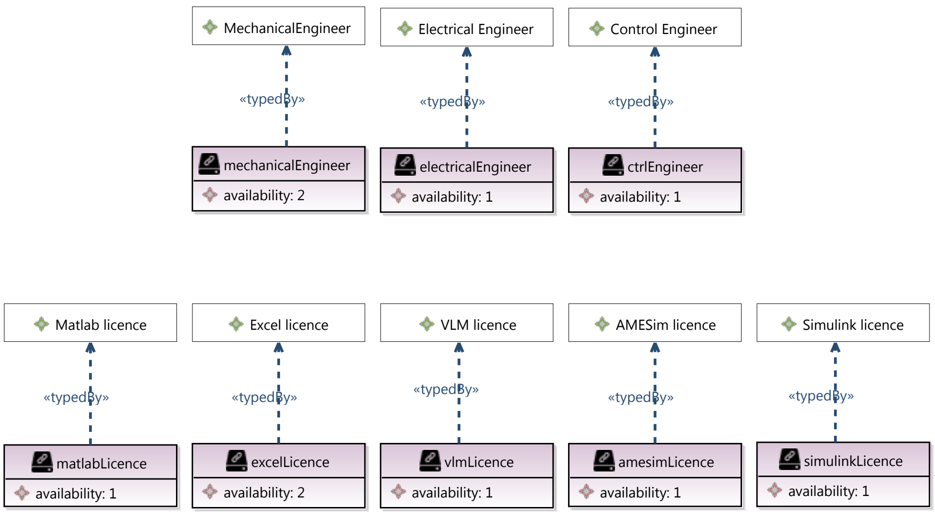


Figure 6.9: The resource model of the demonstrative example.

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 [51].

In order to keep the demonstrative example fairly simple, 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 demonstrative example. 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.

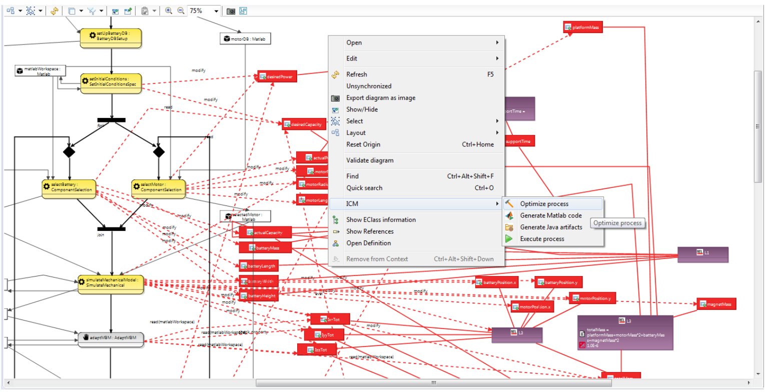


Figure 6.10: Invoking the optimization mechanism on the PM.

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.

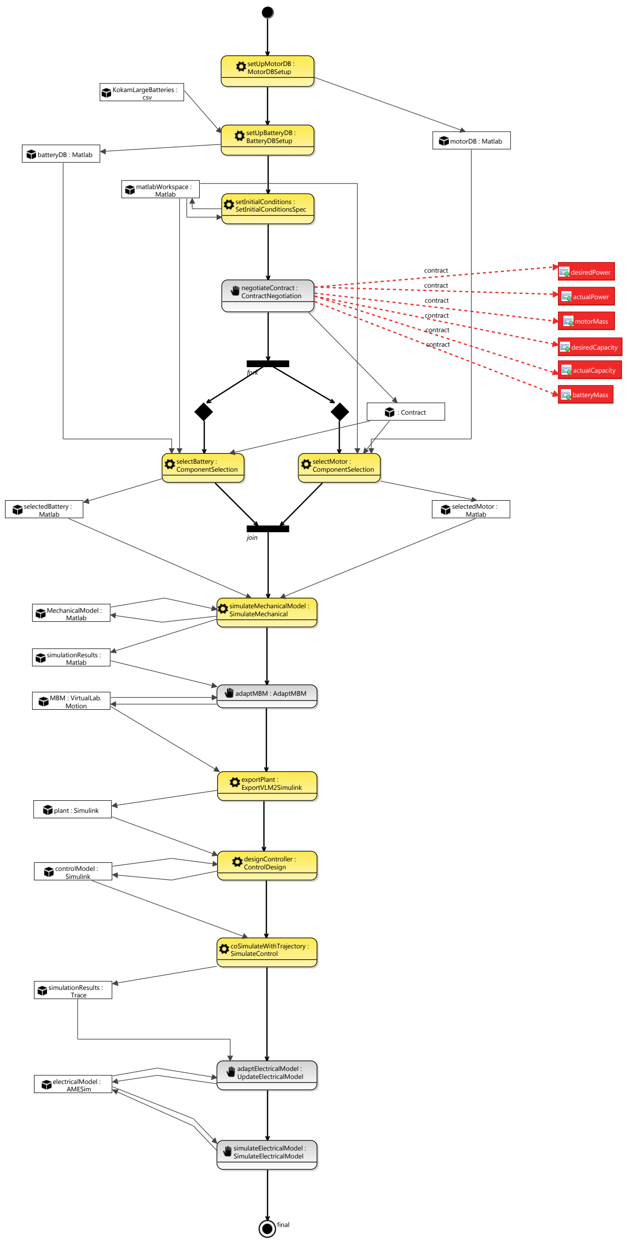

setUpMotorDB : MotorDBSetup
KokamLargeBatteries : csv
setUpBatteryDB : BatteryDBSetup
batteryDB : Matlab
motorDB : Matlab
matlabWorkspace : Matlab
setInitialConditions : SetInitialConditionsSpec
negotiateContract : ContractNegotiation
contract
contract
contract
contract
contract
contract
desiredPower
actualPower
motorMass
desiredCapacity
actualCapacity
batteryMass
fork
: Contract
selectBattery : ComponentSelection
selectMotor : ComponentSelection
selectedBattery : Matlab
selectedMotor : Matlab
join
simulateMechanicalModel : SimulateMechanical
MechanicalModel : Matlab
simulationResults : Matlab
adaptMBM : AdaptMBM
MBM : VirtualLab. Motion
exportPlant : ExportVLM2Simulink
plant : Simulink
designController : ControlDesign
controlModel : Simulink
coSimulateWithTrajectory : SimulateControl
simulationResults : Trace
adaptElectricalModel : UpdateElectricalModel
electricalModel : AMESim
simulateElectricalModel : SimulateElectricalModel
final

Figure 6.11: The optimized PM of the demonstrative example.

The inconsistency manager identified the potential deviation between the pairs of *desired-Capacity-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 [211]. 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.

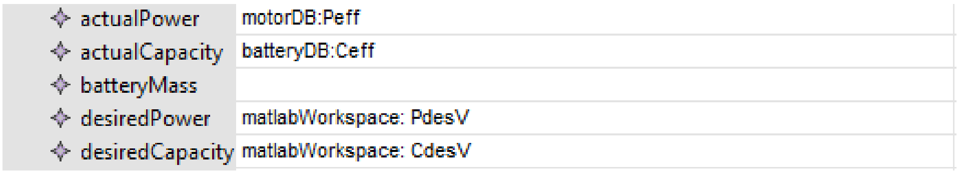
The caveat of this step is obvious: we do not model the internal of the design contract. But this has not been the goal anyways. Modeling design contracts is a research field of its own, and its foundations have been just recently published by Vanherpen [211].

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 [157]. 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 [157] 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.

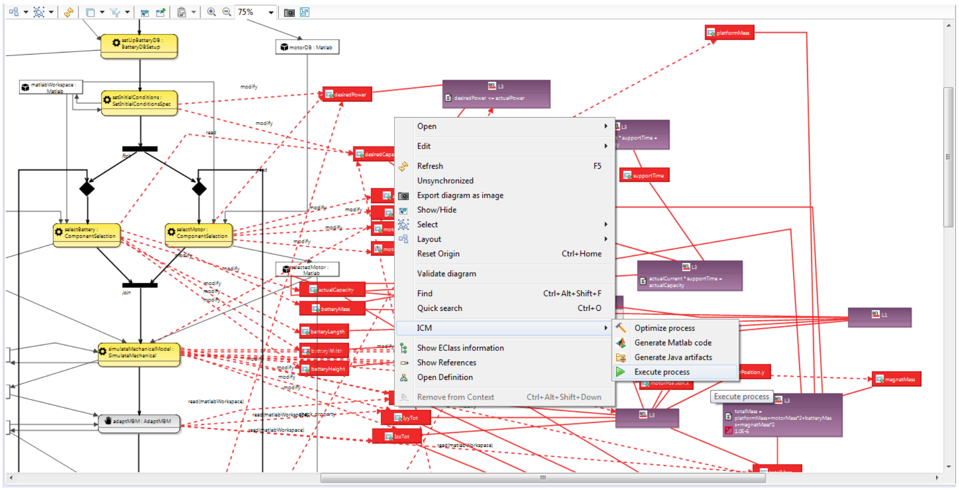


Figure 6.13: Invoking the process enactment mechanism on the PM.

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.

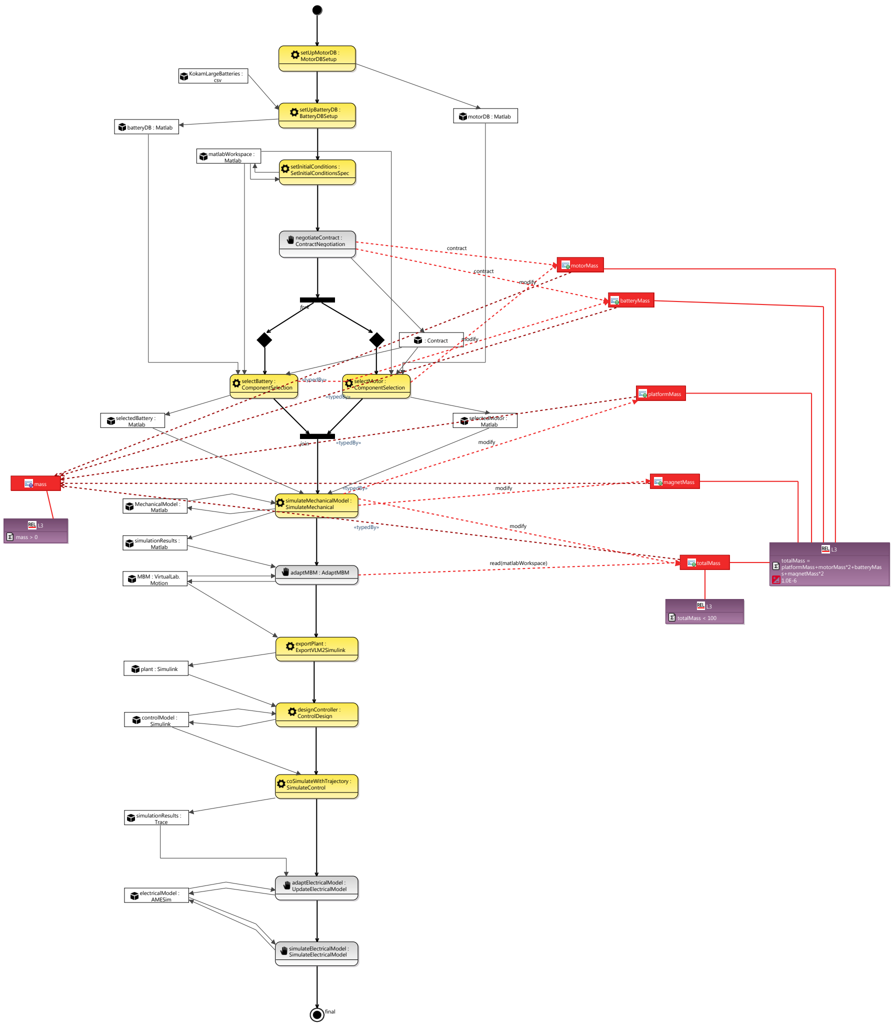


Figure 6.14: The enacted PM with the relevant attributes and capabilities of the on-line inconsistency management phase.

The inconsistency management example shown in Section 5.4 is actually taken from this demonstrative example, thus the reader is referred to that part of this work. 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 work with a prototype modeling and execution tool. The tool is available from http://istvandavid.com/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 *Metamodel*s providing formalisms for both the off-line and the

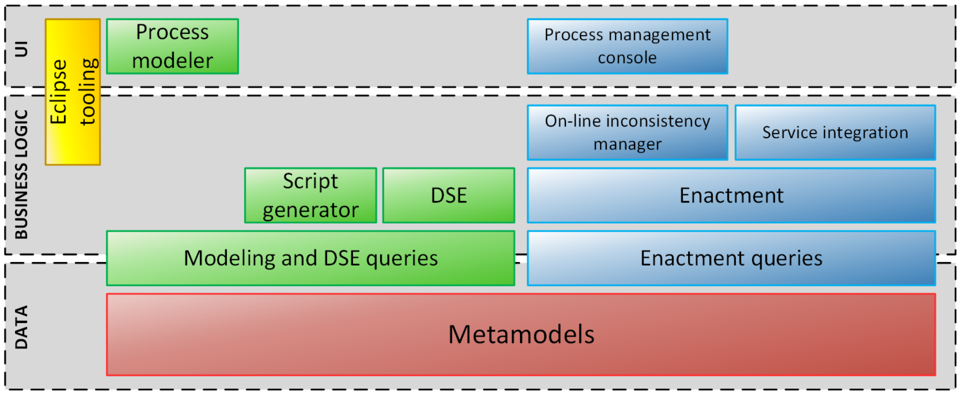


Figure 7.1: General architecture of the tool.

on-line components of the tool. The metamodels used all over the tooling are created using the Eclipse Modeling Framework (EMF) [57]. 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 [125] 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

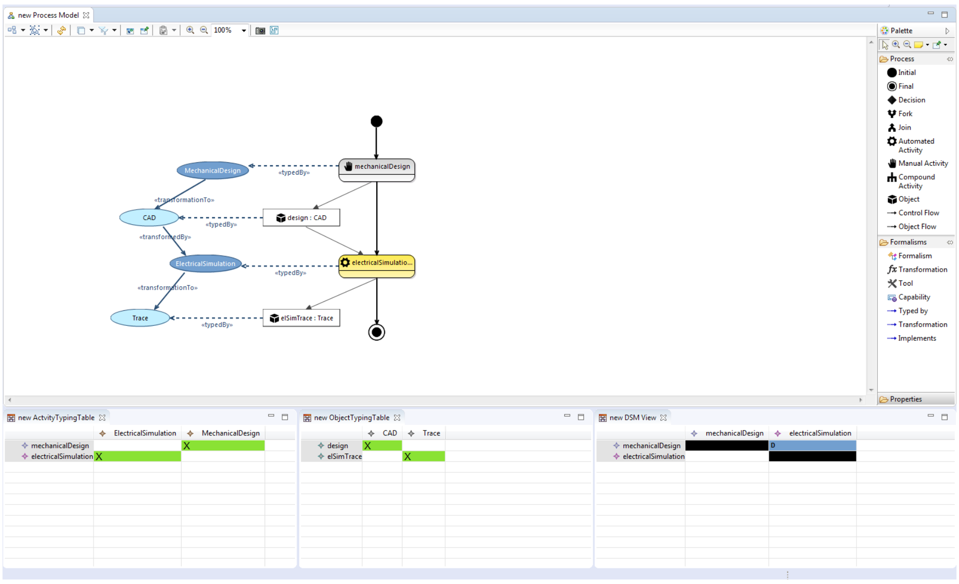
- project nature and project builder contribution,
- a project creation wizard,
- a model creation wizard.

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 [60]. 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.



	ElectricalSimulation	MechanicalDesign
mechanicalDesign		X
electricalSimulation	X	

	CAD	Trace
design	X	
elSimTrace		X

	mechanicalDesign	electricalSimulation
mechanicalDesign	■	D
electricalSimulation		■

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) [65] 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.

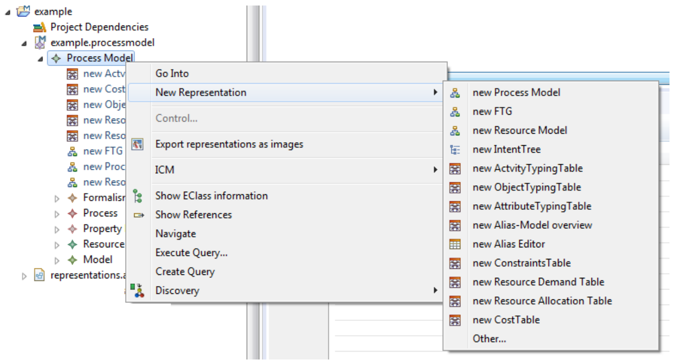


Figure 7.3: Representation types for the *Process Model*.

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 [65].

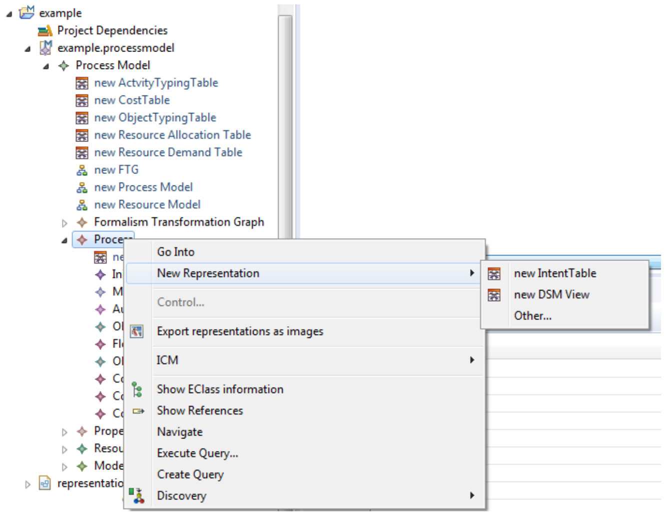


Figure 7.4: Representation types for the PM.

Language elements

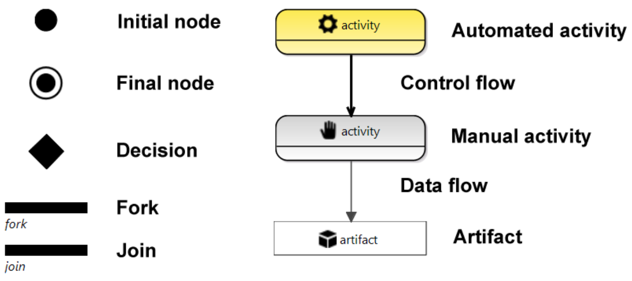


Figure 7.5: Elements for modeling the PM.

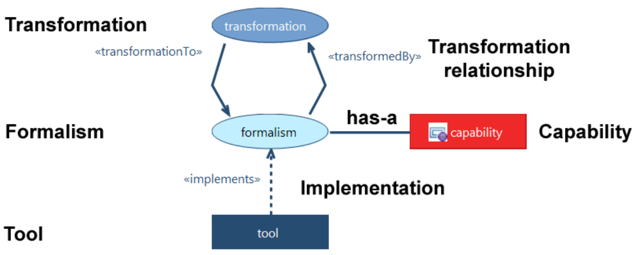


Figure 7.6: Elements for modeling the FTG.

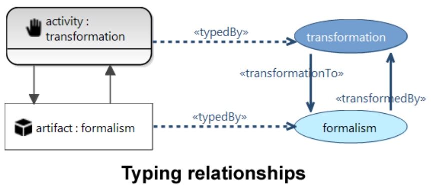


Figure 7.7: Elements for modeling the typing in the FTG+PM.

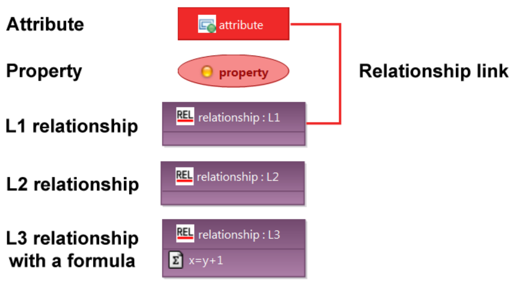


Figure 7.8: Elements for modeling the properties.

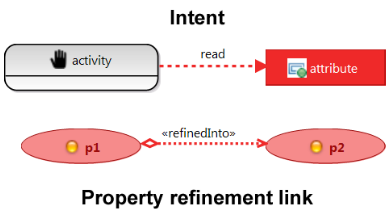


Figure 7.9: Additional elements for modeling the properties.

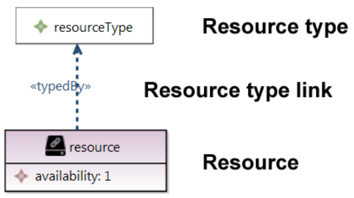


Figure 7.10: Additional elements for modeling the resources.

7.3.2 Eight important modeling patterns

Apart from the process-related patterns [194], there 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.

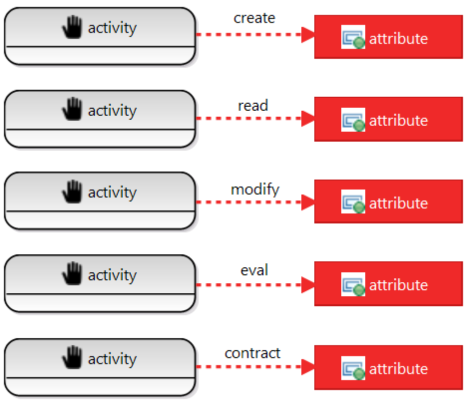


Figure 7.11: Intents.

Model transformation patterns

Here, we follow the taxonomy by Mens and Gorp [129].

Motivation: Expressing the various techniques to transform a model.

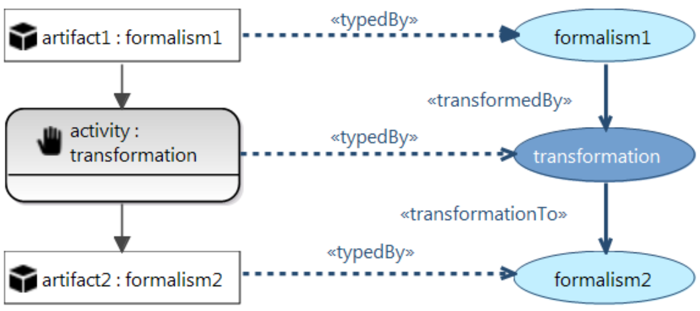


Figure 7.12: Exogenous model transformation.

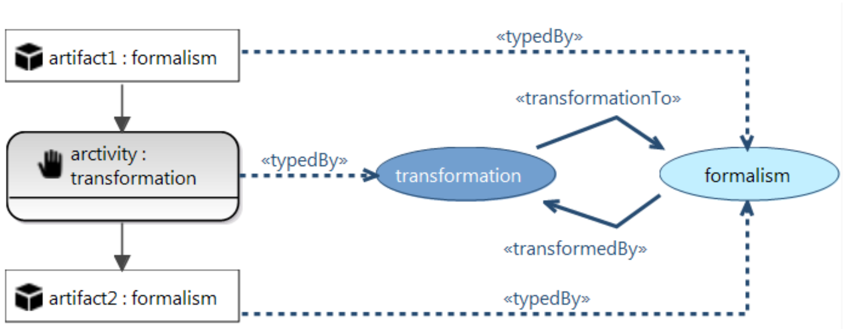


Figure 7.13: Endogenous model transformation.

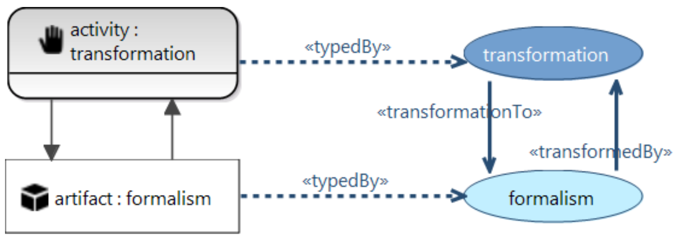


Figure 7.14: Endogenous in-place model transformation.

Property patterns

Here, we follow the taxonomy by Vanherpen et al. [212].

Motivation: Expressing property information, e.g. refinement relationships, or obtaining the performance value of a property.

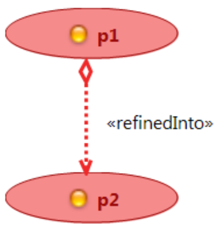


Figure 7.15: Property refinement.

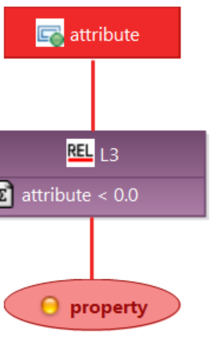


Figure 7.16: Performance value derived through an $\mathbb{L}3$ relationship.

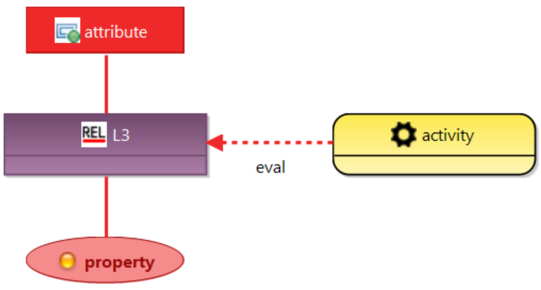


Figure 7.17: Explicit simulation of a performance value.

Capability pattern

Motivation: Expressing the typing relationship between a property and a capability in order to use the latter information in inconsistency detection.

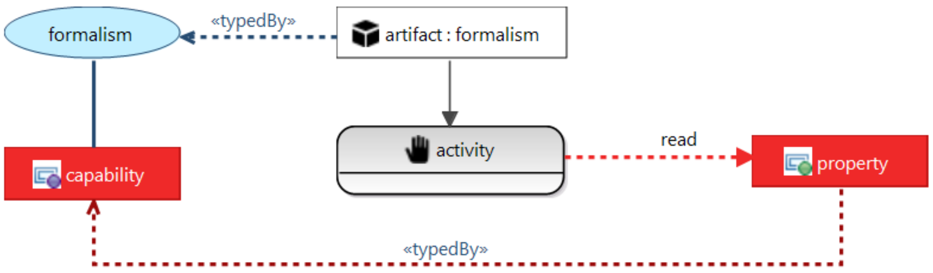


Figure 7.18: Property-capability typing.

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 scrips 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 [28]. 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. [213] for efficient runtime monitoring. Song et al. [175] 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 enatails 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 [183]. 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 [173] is provided. The tool makes use of the managed APIs whenever possible. For Matlab, it's Java API is used [124]. 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 [187] 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 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 [207]. 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 work 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 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. Multi-paradigm modeling advocates modeling everything explicitly, using the most appropriate formalism(s) on the most appropriate level(s) of abstraction [210]. 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 work: (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 work. 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 [210] and M@RT[28] 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 from http://istvandavid.com/icm.

8.1 Evaluation of the research questions

At this point, we reflect on our research questions listed in Chapter 1.

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 [14].

R3. What is an appropriate formalism and 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 work focus on managing inconsistencies on the level of the specific model. UML is often 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 on a low level, i.e., in an intimate proximity to the formalism at hand. Such low-level approaches fail to address some of the macroscopic issues in heterogeneous engineering settings. Our research concluded, that there is a need for the type of inconsistency management techniques that approach inconsistency management on a higher level. We have found that *process modeling formalisms* are especially suitable for this task. The appropriate *level of abstraction* is when the 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 underlying formalism to approach inconsistency management aligns well with the efforts previously made in the fields of multi-paradigm modeling [210] and business process modeling [218].

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 [170, 184, 185, 191].

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 [37]. 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 [190]. 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 work could serve as basis for future researchers, just like many other works 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 [196, 200] are prime candidates to build an approach for such a purpose.

Full translation of process models to DEVS The current approach uses DEVS to provide translational semantics for simulating the performance of the process. DEVS, however, could serve as an assembly language for other modeling languages [209]. Thanks to this versatility, DEVS could be used for multiple purposes in the approach presented in this work.

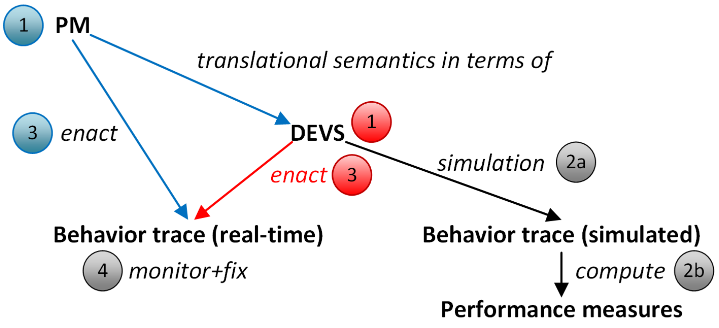

1
PM
translational semantics in terms of
3
enact
DEVS
1
enact
3
simulation
2a
Behavior trace (real-time)
4
monitor+fix
Behavior trace (simulated)
compute
2b
Performance measures

Figure 8.1: DEVS providing semantics for inconsistency management and enactment.

Figure 8.1 shows that the current approach (blue) uses the process semantics to carry out the off-line inconsistency management (*1*, blue); and the same process semantics are used for enactment (*3*, blue). As opposed to this (red), the process could be translated to DEVS immediately and the off-line inconsistency management could be carried out on the DEVS level (*1*, red); furthermore, the DEVS model could be used for enactment as well (*3*, red).

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 [211] 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 [158] shown in this work. 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 [14] is something we have briefly investigated during this research (as discussed in [49]), but eventually fell outside the narrowing scope of the project. Tolerance is a deep topic and deserves to be investigated. 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 [6, 162].

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 [207]; 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 [16].

Appendices

Appendix A

Metamodels

This appendix documents the various parts of the process modeling formalism in greater details.

A.1 Overview

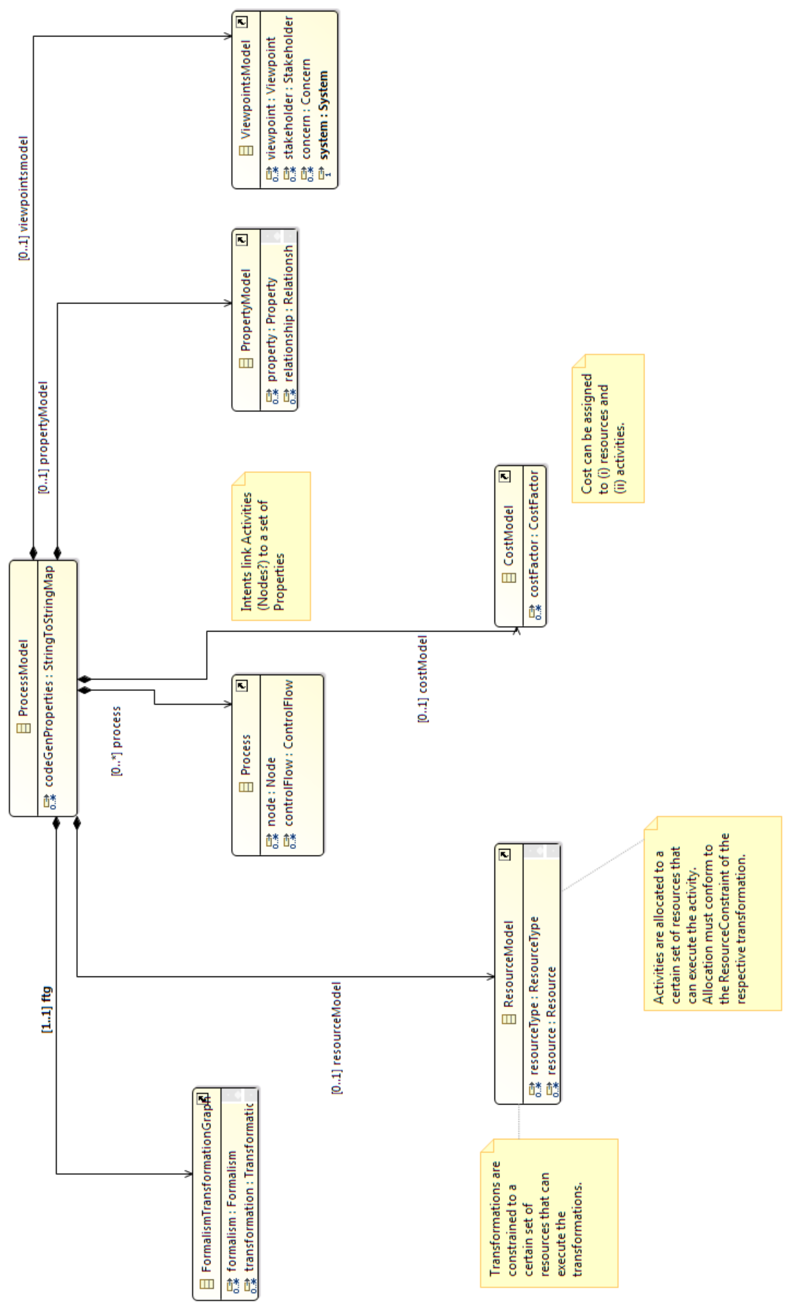

ViewpointsModel
viewpoint : Viewpoint
stakeholder : Stakeholder
concern : Concern
system : System
[0..1] viewpointsmodel
PropertyModel
property : Property
relationship : Relationsh
[0..1] propertyModel
Cost can be assigned to (i) resources and (ii) activities.
Intents link Activities (Nodes?) to a set of Properties
CostModel
costFactor : CostFactor
ProcessModel
codeGenProperties : StringToStringMap
[0..1] costModel
[0..*] process
Process
node : Node
controlFlow : ControlFlow
ResourceModel
resourceType : ResourceType
resource : Resource
Activities are allocated to a certain set of resources that can execute the activity. Allocation must conform to the ResourceConstraint of the respective transformation.
[1..1] ftg
[0..1] resourceModel
FormalismTransformationGraph
formalism : Formalism
transformation : Transformatic
Transformations are constrained to a certain set of resources that can execute the transformations.

Figure A.1: Top-level overview of the formalism.

A.2 FTG+PM

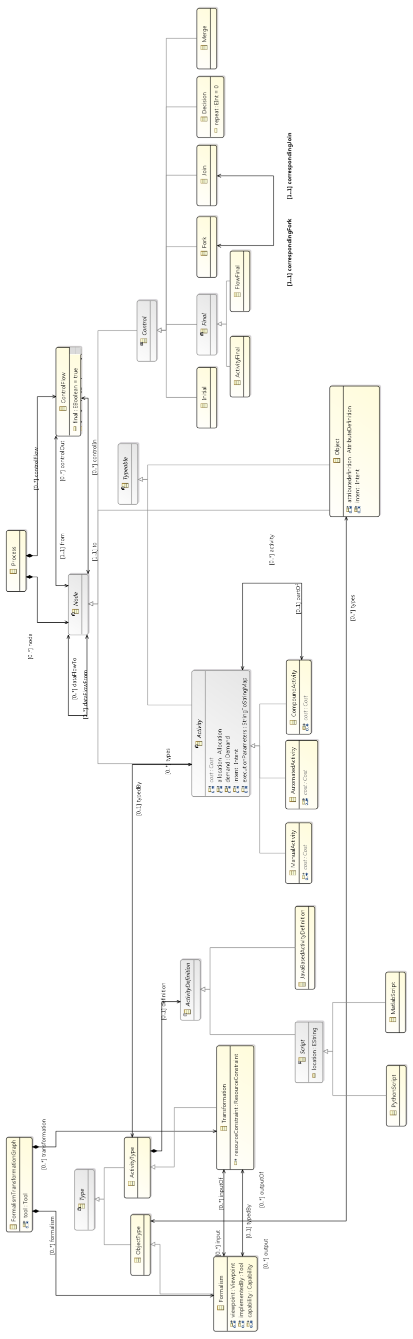


Figure A.2: The FTG+PM modeling part of the formalism.

A.3 Properties

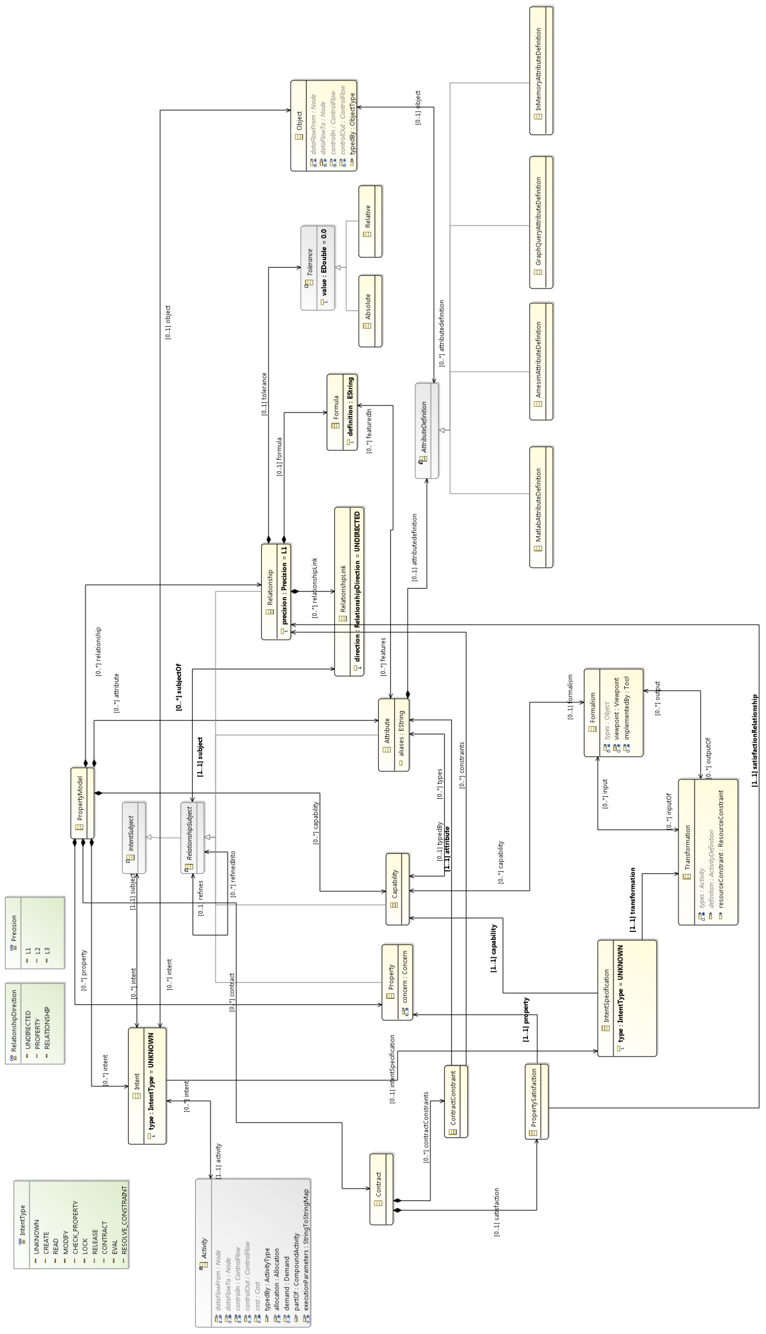


Figure A.3: The property modeling part of the formalism.

A.4 Resources

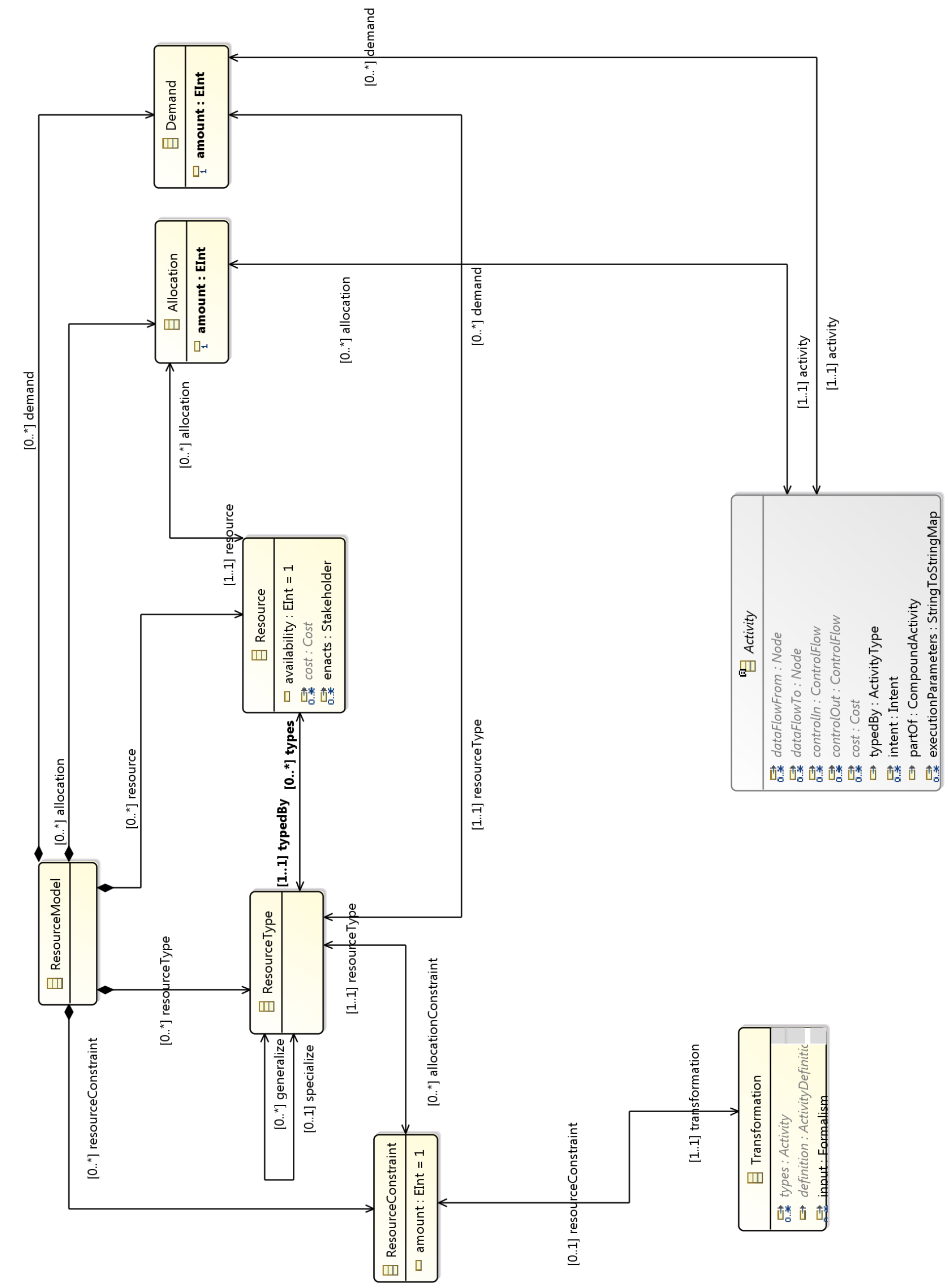


Figure A.4: The resources modeling part of the formalism.

A.5 Costs

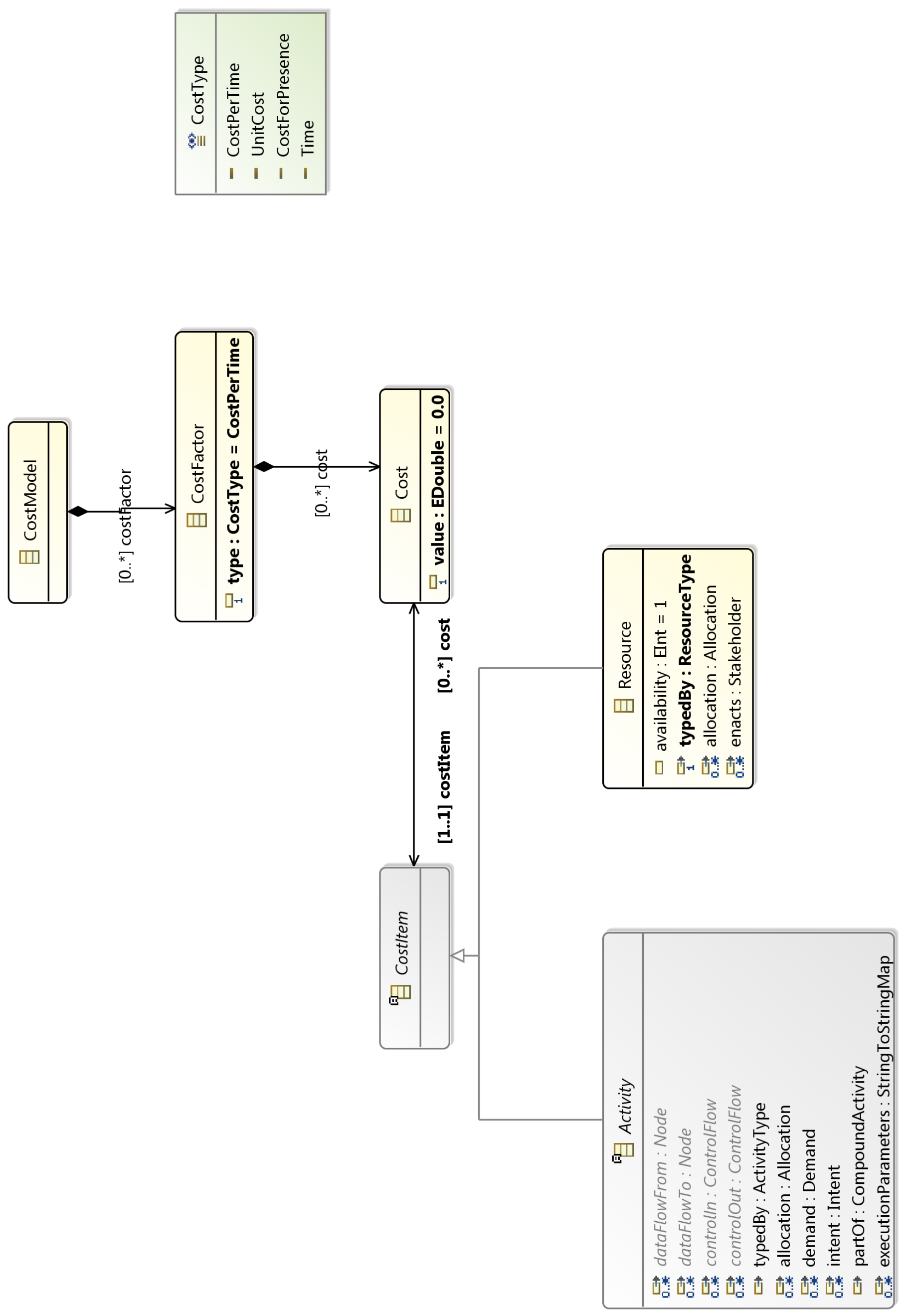


Figure A.5: The cost modeling part of the formalism.

A.6 Viewpoints

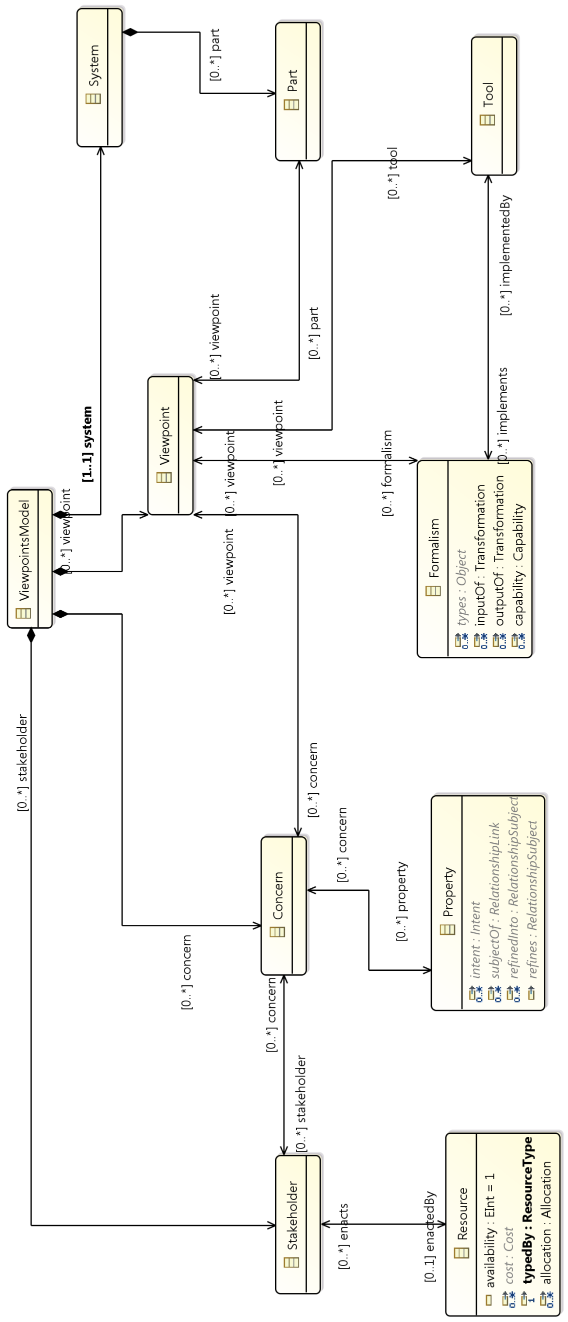


Figure A.6: The viewpoint modeling part of the formalism.

A.7 Enactment

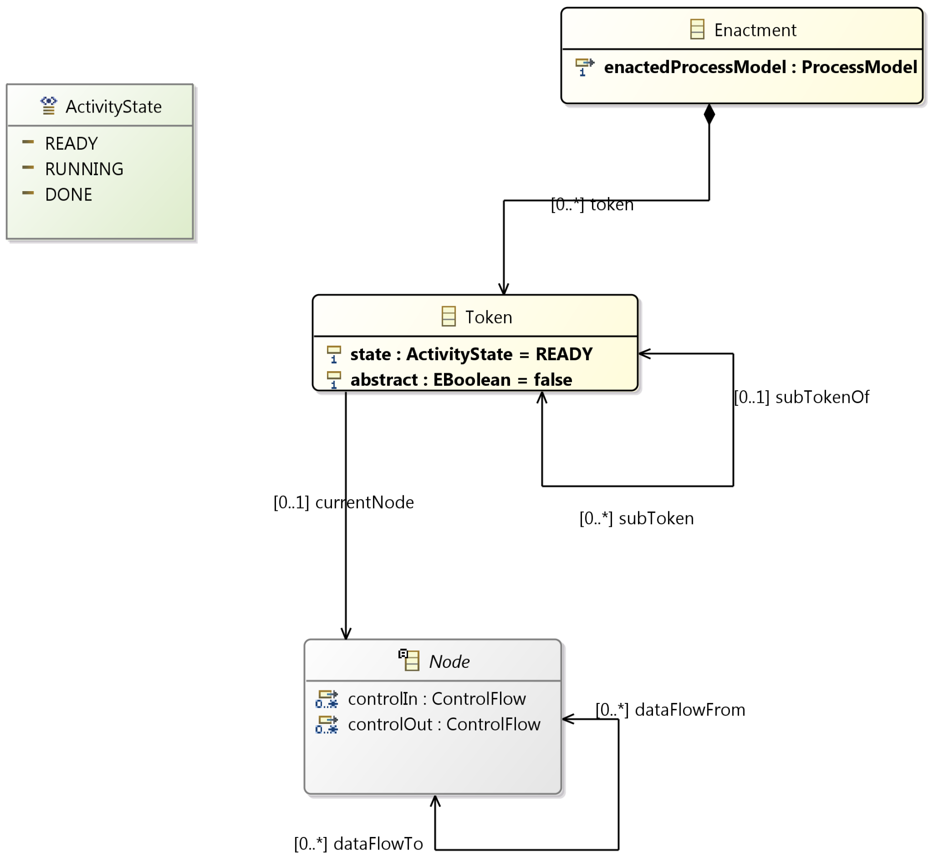


Figure A.7: The enactment modeling part of the formalism.

Appendix B

Model queries and transformations

Listing 5 Patterns of inconsistency.

```
pattern sharedProperty(
  activity1: Activity, intent1: Intent, property1: Property,
  activity2: Activity, intent2: Intent, property2: Property){
    activity1!=activity2;
    find intent(activity1, property1, intent1);
    find intent(activity2, property2, intent2);
    find propertyGloballyReachableFromProperty(property1, property2);
}

pattern sequentialSharedProperty(
  activity1: Activity, intent1: Intent, property1: Property,
  activity2: Activity, intent2: Intent, property2: Property){
    find sharedProperty(
      activity1, intent1, property1, activity2, intent2, property2);
    find nodeGloballyReachableFromNode(activity2, activity1);
}

pattern parallelSharedProperty(
  activity1: Activity, intent1: Intent, property1: Property,
  activity2: Activity, intent2: Intent, property2: Property,
  fork: Fork){
    find sharedProperty(
      activity1, intent1, property1, activity2, intent2, property2);
    find parallelActivities2(activity1, activity2, fork, _);
}

pattern readModifySharedProperty(
  activity1: Activity, property1: Property,
  activity2: Activity, property2: Property){
    find sequentialSharedProperty(
      activity1, intent1, property1, activity2, intent2, property2);
    Intent.type(intent1, IntentType::READ);
    Intent.type(intent2, IntentType::MODIFY);
}

pattern modifyModifySharedPropertySequential(
  activity1: Activity, property1: Property,
  activity2: Activity, property2: Property){
    find sequentialSharedProperty(
      activity1, intent1, property1, activity2, intent2, property2);
    Intent.type(intent1, IntentType::MODIFY);
    Intent.type(intent2, IntentType::MODIFY);
}

pattern modifyModifySharedPropertyParallel(
  activity1: Activity, property1: Property,
  activity2: Activity, property2: Property,
  fork: Fork){
    find parallelSharedProperty(
    activity1, intent1, property1, activity2, intent2, property2, fork);
    Intent.type(intent1, IntentType::MODIFY);
    Intent.type(intent2, IntentType::MODIFY);
}
```

Listing 6 Patterns resource allocation.

```
pattern allocatedActivity(
  activity: Activity, allocation: Allocation, resource: Resource){
    neg find compoundActivity(activity);
    Activity.allocation(activity, allocation);
    Allocation.resource(allocation, resource);
}

pattern unAllocatedActivity(activity: Activity){
  neg find compoundActivity(activity);
  neg find allocatedActivity(activity, _, _);
}

pattern demand(activity: Activity, demand: Demand, resourceType: ResourceType){
  neg find compoundActivity(activity);
  Activity.demand(activity, demand);
  Demand.resourceType(demand, resourceType);
}

pattern resourceInstance(resource: Resource, resourceType: ResourceType){
  Resource.typedBy(resource, resourceType);
}

pattern unsatisfiedDemand(activity: Activity){
  neg find compoundActivity(activity);

  find demand(activity, demand, resourceType);
  find allocatedActivity(activity, allocation, resource);

  find resourceInstance(resource, resourceType);

  Demand.amount(demand, dAmount);
  Allocation.amount(allocation, aAmount);

  check(aAmount < dAmount);
}

pattern satisfiedDemand(activity: Activity){
  neg find compoundActivity(activity);
  neg find unsatisfiedDemand(activity);
}

pattern satisfiableDemand(activity: Activity, resourceType: ResourceType){
  neg find compoundActivity(activity);

  find demand(activity, demand, resourceType);

  find resourceInstance(resource, resourceType);

  Demand.amount(demand, dAmount);
  Resource.availability(resource, availability);

  check(availability >= dAmount);
}

pattern unsatisfiableDemand(activity: Activity, resourceType: ResourceType){
  neg find compoundActivity(activity);

  find demand(activity, demand, resourceType);

  find resourceInstance(resource, resourceType);

  Demand.amount(demand, dAmount);
  Resource.availability(resource, availability);

  check(availability < dAmount);
}
```

Listing 7 Management transformation rules for the sequential read-modify inconsistency pattern. Reuses patterns from Listing 5.

```
/**
* Reordering
*/
val readModifyReorder = new DSETransformationRule(
  unmanagedReadModify,
  new UnmanagedReadModifyProcessor() {
    override process(Activity activity1, Property property1,
      Activity activity2, Property property2) {
        val tmp = createManualActivity("tmp");

        tmp.controlIn.addAll(activity1.controlIn)
        activity1.controlIn.removeAll(tmp.controlIn)

        tmp.controlOut.addAll(activity1.controlOut)
        activity1.controlOut.removeAll(tmp.controlOut)

        activity1.controlIn.addAll(activity2.controlIn)
        activity2.controlIn.removeAll(activity1.controlIn)

        activity1.controlOut.addAll(activity2.controlOut)
        activity2.controlOut.removeAll(activity1.controlOut)

        activity2.controlIn.addAll(tmp.controlIn)
        tmp.controlIn.removeAll(activity2.controlIn)

        activity2.controlOut.addAll(tmp.controlOut)
        tmp.controlOut.removeAll(activity2.controlOut)
      }
  }
)

/**
 * Check property
 */
val readModifyAugmentWithCheck = new DSETransformationRule(
  unmanagedReadModify2,
  new UnmanagedReadModify2Processor() {
    override process(Activity activity1, Property property1,
      Activity activity2, Property property2) {
        createDecision(activity2, property1, activity1)
      }
  }
)

val readModifyAugmentWithContract = new DSETransformationRule(
  unmanagedReadModify3,
  new UnmanagedReadModify3Processor() {
    override process(Activity activity1, Property property1,
      Activity activity2, Property property2) {
        createContract(activity1, #[property1], activity1)
    }
  }
)

val modifyModifyAugmentWithContract = new DSETransformationRule(
  unmanagedModifyModifySequential,
  new UnmanagedModifyModifySequentialProcessor() {
    override process(Activity activity1, Property property1,
      Activity activity2, Property property2) {
        createContract(activity1, #[property1], activity1)
    }
  }
)
```

Listing 8 Transformation rule of the contract-based management pattern for the for the parallel modify-modify inconsistency pattern. Reuses patterns from Listing 5.

```
val addContract = new DSETransformationRule(
  unmanagedModifyModifyParallel,
  new UnmanagedModifyModifyParallelProcessor() {
    override process(Activity activity1, Property property1,
      Activity activity2, Property property2, Fork fork) {
        createContract(fork, #[property1, property2], activity1)
    }
  }
)
```

Listing 9 Soft and hard objectives for the process space exploration.

```
val consistencyObjective =
  new ConstraintsObjective()
  .withSoftConstraint("consistencyObjective",unmanagedReadModify, 10d)
  .withComparator(Comparators.LOWER_IS_BETTER)

val cheapestProcessObjective =
  new CheapestProcessSoftObjective()
  .withComparator(Comparators.LOWER_IS_BETTER)

val allocationObjectives =
  new ConstraintsObjective("validAllocation")
  .withHardConstraint(unAllocatedActivity, ModelQueryType::NO_MATCH)
  .withLevel(0)

val validationObjectives =
  new ConstraintsObjective("validProcess")
    .withHardConstraint(
      initNodeWithInvalidNumberOfControlOut, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      initNodeWithControlIn, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      finalNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      finalNodeWithControlOut, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      forkNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      forkNodeWithInvalidNumberOfOuts, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      joinNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      joinNodeWithInvalidNumberOfOuts, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      decisionNodeWithInvalidNumberOfIns, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      decisionNodeWithInvalidNumberOfOuts, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      activityWithInvalidNumberOfControlIn, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      activityWithInvalidNumberOfControlOut, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      controlFlowWithInvalidNumberOfControlFrom, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      controlFlowWithInvalidNumberOfControlTo, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      redundantControlFlows,ModelQueryType::NO_MATCH)
    .withHardConstraint(
      finalNotReachableFromNode, ModelQueryType::NO_MATCH)
    .withHardConstraint(
      initDoesNotReachNode, ModelQueryType::NO_MATCH)
    .withLevel(0)
```

Listing 10 Model patterns used in the process enactment runtime - Part 1.

```
pattern activeToken(token: Token){
  Token.abstract(token, a);
  a == false;
}

pattern tokenInNode(token : Token, node : Node){
  find activeToken(token);
  Token.currentNode(token, node);
}

pattern tokenInActivity(token : Token, node : Activity){
  find activeToken(token);
  find tokenInNode(token, node);
}

pattern activity(node: Node){
  Activity(node);
}

pattern attributeModificationActivity(
  activity: Activity, attribute: Attribute){
    Intent.activity(intent, activity);
    Intent.subject(intent, attribute);
    Intent.type(intent, ::MODIFY);
}

pattern fireable(
  token: Token, controlFlow: ControlFlow, fromNode: Node, toNode: Node){
  //fire from a non-activity
  find tokenInNode(token, fromNode);
  ControlFlow.from(controlFlow, fromNode);
  ControlFlow.to(controlFlow, toNode);
  neg find activity(fromNode);
  neg find splitTokenInJoin(token, fromNode);
}
or{
  //fire from a non-activity to a node where
  //the direct sibling of the token is located
  find tokenInNode(token, fromNode);
  ControlFlow.from(controlFlow, fromNode);
  ControlFlow.to(controlFlow, toNode);
  neg find activity(fromNode);
  find splitTokenInJoin(token, fromNode);
  find splitTokenInJoin(token2, toNode);
  find directSiblingTokens(token, token2);
}
or{
  //fire from an activity
  find tokenInNode(token, fromNode);
  ControlFlow.from(controlFlow, fromNode);
  ControlFlow.to(controlFlow, toNode);
  find doneActivity(token, fromNode);
}

pattern availableActivity(
  token: Token, controlFlow: ControlFlow, activity: Activity){
  find fireable(token, controlFlow, _, activity);
}

pattern availableFinish(token: Token, controlFlow: ControlFlow, final: Final){
  find fireable(token, controlFlow, _, final);
}
```

Listing 11 Model patterns used in the process enactment runtime - Part 2. (Continuation of Listing 10.)

```
pattern fireableToControl(token: Token, fromNode: Node, control: Control){
  find fireable(token, _, fromNode, control);
  Fork(control);
}or{
  find fireable(token, _, fromNode, control);
  Join(control);
}or{
  find fireable(token, _, fromNode, control);
  Decision(control);
}or{
  find fireable(token, _, fromNode, control);
  Merge(control);
}

pattern splitTokenInJoin(token: Token, join: Join){
  find tokenInNode(token, join);
  find siblingTokens(token, token2);
  find tokensInDifferentNodes(token, token2);
}

pattern joinable(join: Join){
  find tokenInNode(token, join);
  find siblingTokens(token, token2);
  neg find tokensInDifferentNodes(token, token2);
}

pattern forkable(fork: Fork, token: Token){
  find tokenInNode(token, fork);
}

pattern subTokenOf(subToken: Token, parent: Token){
  Token.subTokenOf(subToken, parent);
}

pattern siblingTokens(token1: Token, token2: Token){
  find activeToken(token1);
  find activeToken(token2);
  find subTokenOf+(token1, parent);
  find subTokenOf+(token2, parent);
  token1 != token2;
}

pattern directSiblingTokens(token1: Token, token2: Token){
  find activeToken(token1);
  find activeToken(token2);
  find subTokenOf(token1, parent);
  find subTokenOf(token2, parent);
  token1 != token2;
}

pattern tokensInDifferentNodes(token1: Token, token2: Token){
  Token.currentNode(token1, node1);
  Token.currentNode(token2, node2);
  node1 != node2;
}

//to detect runnable activity
pattern readyActivity(token : Token, node : Activity){
  find tokenInActivity(token, node);
  Token.state(token, ::READY);
}
```

Listing 12 Model patterns used in the process enactment runtime - Part 3. (Continuation of Listing 11.)

```
//to detect running activity
pattern runnigActivity(token : Token, node : Activity){
  find tokenInActivity(token, node);
  Token.state(token, ::RUNNING);
}

//to detect executed activity
pattern doneActivity(token : Token, node : Activity){
  find tokenInActivity(token, node);
  Token.state(token, ::DONE);
}

//to split tokens
pattern tokenInFork(token : Token, node : Fork){
  find tokenInNode(token, node);
}

//to re-join tokens
pattern tokenInJoin(token : Token, node : Join){
  find tokenInNode(token, node);
}

//to multiplex tokens
pattern tokenInDecision(token : Token, node : Decision){
  find tokenInNode(token, node);
}

//to de-multiplex tokens
pattern tokenInMerge(token : Token, node : Merge){
  find tokenInNode(token, node);
}

//to designate the end of an execution
pattern finishedProcess(token : Token, node : FlowFinal){
  find tokenInNode(token, node);
}

//to enable moving tokens from one node to another
pattern enabledTransition(fromNode : Node, toNode : Node, token: Token){
  ControlFlow.from(controlFlow, fromNode);
  ControlFlow.to(controlFlow, toNode);
  find tokenInNode(token, fromNode);
  Token.state(token, ::DONE);
}
```

Listing 13 Elementary model transformations used in the process enactment runtime. Reuses the patterns from Listing 10, Listing 11 and and Listing 12.

```
val fireToControlRule = createRule.name("fire to control")
  .precondition(fireableToControl)
  .action [
    logger.debug(String.format(
      "Firing token %s from node %s to control node %s.",
      token, fromNode, control))
    token.currentNode = control
  ]
  .build

val forkableRule = createRule.name("forkable")
  .precondition(forkable)
  .action [
    logger.debug(String.format("Forking token %s at %s.", token, fork))
    // de-activate parent
    token.abstract = true

    // create sub-tokens
    for (ctrlOut : fork.controlOut) {
      val newToken = EnactmentFactory.eINSTANCE.createToken
      newToken.subTokenOf = token
      enactment.token.add(newToken)
      newToken.currentNode = ctrlOut.to
      if (ctrlOut.to instanceof Activity) {
        newToken.state = ActivityState::READY
      }
    }
    println("forkable")
  ]
  .build

val joinableRule = createRule.name("joinable")
  .precondition(joinable)
  .action [
    logger.debug(String.format("Joining tokens at %s.", join))
    val tokenMatches = queryEngine
      .getMatcher(TokenInNodeQuerySpecification.instance)
      .allMatches.filter [ match | match.node.equals(join)
    ] // each token at this point should be joinable
    logger
    .debug(String.format("Joinable tokens: %s.",
      tokenMatches.map[tm|tm.token].toList))

    // activate parent
    val parentToken = tokenMatches.head.token.subTokenOf
    parentToken.abstract = false
    parentToken.currentNode = join

    // remove subs
    for (tokenMatch : tokenMatches) {
      enactment.token.remove(tokenMatch.token)
    }
    println("joinable")
  ].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.

```
from pypdevs.DEVS import *

class ResourceHandler(AtomicDEVS):
    def __init__(self, resources):
        AtomicDEVS.__init__(self, "__resource_handler")
        self.state = {'resources': resources, 'queue': []}
        self.elapsed = 0.0
        self.resource_in = self.addInPort("resource_in")
        self.resource_out = self.addOutPort("resource_out")

    def extTransition(self, inputs):
        for inp in inputs[self.resource_in]:
            if inp['type'] == "request":
                # Queue the request
                self.state['queue'].append(inp['id'])
            elif inp['type'] == "release":
                # Increment the number of available resources
                self.state['resources'] += 1

        return self.state

    def intTransition(self):
        # Processed a request that could be processed
        self.state['resources'] -= 1
        del self.state['queue'][0]
        return self.state

    def outputFnc(self):
        return {self.resource_out: {'id': self.state['queue'][0]}}

    def timeAdvance(self):
        if self.state['queue'] and self.state['resources']:
            # Can grant a resource
            return 0.0
        else:
            # No request queued, or no available resources to handle the request
            return float('inf')
```

Listing 15 Atomic DEVS model library for processes - Part 2.

```python
class ActivityState(object):
    def __init__(self, name, distribution):
        self.timer = float('inf')
        self.mode = "inactive"
        self.counter = 0
        self.name = name
        self.distribution = distribution

    def __str__(self):
        return str(vars(self))

    def random_sample(self):
        return self.distribution(self.counter)

class Activity(AtomicDEVS):
    def __init__(self, name, distribution):
        AtomicDEVS.__init__(self, name)
        self.state = ActivityState(name, distribution)
        self.elapsed = 0.0
        self.control_in = self.addInPort("control_in")
        self.resource_in = self.addInPort("resource_in")
        self.control_out = self.addOutPort("control_out")
        self.resource_out = self.addOutPort("resource_out")

    def intTransition(self):
        self.state.timer -= self.timeAdvance()
        if self.state.mode == "request_resource":
            # Go and request the required resource
            self.state.mode = "wait_resource"
            self.state.timer = float('inf')
        elif self.state.mode == "active":
            # Finished execution, so release resources
            self.state.mode = "inactive"
            self.state.timer = 0.0
            self.state.timer = float('inf')
            self.state.counter += 1
        return self.state

    def extTransition(self, inputs):
        self.state.timer -= self.elapsed
        if self.state.mode == "inactive" and self.control_in in inputs:
            # Got control token, so ask for the required resources
            self.state.mode = "request_resource"
            self.state.timer = 0.0
            # NOTE this violates DEVS, though is easy to debug
            #print("Activate " + str(self.state.name) + " at time " + str(self.time_last[0] + self.elapsed))
        elif self.state.mode == "wait_resource" and self.resource_in in inputs and inputs[self.resource_in]['id'] == "%s-%s" % (self.state.name, self.state.counter):
            # Got required resources, so start execution
            self.state.mode = "active"
            self.state.timer = self.state.random_sample()
        return self.state

    def timeAdvance(self):
        return self.state.timer

    def outputFnc(self):
        if self.state.mode == "active":
            # Output the control token to the next model in line, and release the resources
            return {self.control_out: {}, self.resource_out: [{'type': "release"}]}
        elif self.state.mode == "request_resource":
            # Output a request for resources with a specified ID (used to find out whether this was our request)
            return {self.resource_out: [{'type': "request", 'id': "%s-%s" % (self.state.name, self.state.counter)}]}
        else:
            return {}
```

Listing 16 Atomic DEVS model library for processes - Part 3.

```python
class ParallelSplit(AtomicDEVS):
    def __init__(self, name):
        AtomicDEVS.__init__(self, name)
        self.control_in = self.addInPort("control_in")
        self.control_out = self.addOutPort("control_out")
        self.state = False

    def intTransition(self):
        return False

    def extTransition(self, inputs):
        return True

    def outputFnc(self):
        return {self.control_out: {}}

    def timeAdvance(self):
        if self.state:
            return 0.0
        else:
            return float('inf')

class Synchronization(AtomicDEVS):
    def __init__(self, name, counter):
        AtomicDEVS.__init__(self, name)
        self.state = {'current': counter, 'max': counter}
        self.control_in = self.addInPort("control_in")
        self.control_out = self.addOutPort("control_out")

    def intTransition(self):
        self.state['current'] = self.state['max']
        return self.state

    def extTransition(self, inputs):
        self.state['current'] -= 1
        return self.state

    def timeAdvance(self):
        if self.state['current'] == 0:
            return 0.0
        else:
            return float('inf')

    def outputFnc(self):
        return {self.control_out: {}}

class ExclusiveChoiceState(object):
    def __init__(self, outputs, distribution):
        self.counter = 0
        self.outputs = outputs
        self.choice = None
        self.distribution = distribution

    def __str__(self):
        return str(vars(self))

    def make_choice(self):
        return self.distribution(self.counter)
```

Listing 17 Atomic DEVS model library for processes - Part 4.

```
class ExclusiveChoice(AtomicDEVS):
    def __init__(self, name, outputs, distribution):
        AtomicDEVS.__init__(self, name)
        self.state = ExclusiveChoiceState(outputs, distribution)
        self.control_in = self.addInPort("control_in")
        self.control_out = [self.addOutPort("control_out_%s" % i) for i in range(
             ↪ outputs)]

    def intTransition(self):
        self.state.choice = None
        self.state.counter += 1
        return self.state

    def extTransition(self, inputs):
        # Got a control token, so have to make a choice
        self.state.choice = self.state.make_choice()
        return self.state

    def outputFnc(self):
        return {self.control_out[self.state.choice]: {}}

    def timeAdvance(self):
        if self.state.choice is not None:
            return 0.0
        else:
            return float('inf')

class SimpleMerge(AtomicDEVS):
    def __init__(self, name):
        AtomicDEVS.__init__(self, name)
        self.state = False
        self.control_in = self.addInPort("control_in")
        self.control_out = self.addOutPort("control_out")

    def intTransition(self):
        return False

    def extTransition(self, inputs):
        return True

    def outputFnc(self):
        return {self.control_out: {}}

    def timeAdvance(self):
        if self.state:
            return 0.0
        else:
            return float('inf')

class MultiInstanceState(object):
    def __init__(self, name, num, distribution):
        self.spawned = num
        self.collected = 0
        self.counter = 0
        self.mode = "inactive"
        self.running_tasks = []
        self.requested = []
        self.name = name
        self.distribution = distribution

    def __str__(self):
        return str(vars(self))

    def task_time(self):
        return self.distribution(self.counter)
```

Listing 18 Atomic DEVS model library for processes - Part 5.

```python
class MultiInstance(AtomicDEVS):
    def __init__(self, name, num, distribution):
        AtomicDEVS.__init__(self, name)
        self.state = MultiInstanceState(name, num, distribution)
        self.control_in = self.addInPort("control_in")
        self.resource_in = self.addInPort("resource_in")
        self.control_out = self.addOutPort("control_out")
        self.resource_out = self.addOutPort("resource_out")

    def intTransition(self):
        ta = self.timeAdvance()
        for t in self.state.running_tasks:
            t[0] -= ta

        if self.state.mode == "active":
            # Finished an instance, so pop it
            del self.state.running_tasks[0]
            self.state.collected += 1
            if self.state.collected == self.state.spawned:
                self.state.mode = "finish"
        elif self.state.mode == "request_resources":
            # Requested resources, so be ready for a response
            self.state.mode = "active"
        elif self.state.mode == "finish":
            self.state.mode = "inactive"
            self.state.collected = 0
            self.state.counter += 1
            self.state.running_tasks = []
        return self.state

    def extTransition(self, inputs):
        # Got input, so have to spawn #num of them
        for t in self.state.running_tasks:
            t[0] -= self.elapsed

        if self.state.mode == "inactive" and self.control_in in inputs:
            self.state.mode = "request_resources"
            self.state.requested = ["%s-%s-%s" % (self.state.name, self.state.counter
                ↪ , i) for i in range(self.state.spawned)]

        if self.state.mode in ["active", "release_resource"] and self.resource_in in
            ↪ inputs:
            # Got a resource, so allocate it to an activity
            self.state.running_tasks.append([self.state.task_time(), self.state.
                ↪ requested.pop(0)])
            # NOTE this violates DEVS, though is easy to debug
            #print("Spawn " + str(self.state.running_tasks[-1][1]) + " at time " +
                ↪ str(self.time_last[0] + self.elapsed))
            self.state.running_tasks.sort()
        return self.state

    def outputFnc(self):
        if self.state.mode == "request_resources":
            # Request all resources in one go
            return {self.resource_out: [{'type': 'request', 'id': i} for i in self.
                ↪ state.requested]}
        elif self.state.mode == "active":
            # Finished an instance, so release it
            return {self.resource_out: [{'type': 'release'}]}
        elif self.state.mode == "finish":
            # Finished execution of all, so pass on token
            return {self.control_out: {}}
        else:
            return {}

    def timeAdvance(self):
        if self.state.mode == "finish":
            return 0.0
        elif self.state.running_tasks:
            return self.state.running_tasks[0][0]
        elif self.state.mode == "request_resources":
            return 0.0
        else:
            return float('inf')
```

Listing 19 Atomic DEVS model library for processes - Part 6.

```
class Initial(AtomicDEVS):
    def __init__(self):
        AtomicDEVS.__init__(self, "Initial")
        self.control_out = self.addOutPort("control_out")
        self.state = True

    def intTransition(self):
        return False

    def outputFnc(self):
        return {self.control_out: {}}

    def timeAdvance(self):
        if self.state:
            return 0.0
        else:
            return float('inf')

class Finish(AtomicDEVS):
    def __init__(self):
        AtomicDEVS.__init__(self, "Finish")
        self.control_in = self.addInPort("control_out")
        self.state = None

    def extTransition(self, inputs):
        return self.elapsed

    def timeAdvance(self):
        return float('inf')
```

Listing 20 Coupled DEVS model of the running example - Part 1.

```python
from library import *
from pypdevs.simulator import Simulator
from pypdevs.DEVS import CoupledDEVS
import random
import math

def df(iteration):
    result = random.random() < {0: 0.99, 1: 0.9, 2: 0.8, 3: 0.5, 4: 0.2, 5: 0.1, 6:
        ↪ 0.02}.get(iteration, 0.0)
    return int(result)

def cf(mean):
    def cf_int(iteration):
        mu = mean * (1-math.exp(-(iteration+1) / 0.7))
        sigma = mu*0.15625
        return max(0.0, random.gauss(mu, sigma))
    return cf_int

class Example(CoupledDEVS):
    def __init__(self, nresources):
        CoupledDEVS.__init__(self, "example")
        initial = self.addSubModel(Initial())
        req = self.addSubModel(Activity("define_requirements", cf(10.0)))
        merge = self.addSubModel(SimpleMerge("merge"))
        do_model = self.addSubModel(Activity("update_model", cf(100.0)))
        split = self.addSubModel(ParallelSplit("split"))
        sim = self.addSubModel(MultiInstance("simulation", 10, cf(20.0)))
        check = self.addSubModel(Activity("checking", cf(30.0)))
        synchronization = self.addSubModel(Synchronization("synchronization", 2))
        evaluation = self.addSubModel(Activity("evaluation", cf(0.1)))
        choice = self.addSubModel(ExclusiveChoice("choice", 2, df))
        self.finish = self.addSubModel(Finish())

        resources = self.addSubModel(ResourceHandler(nresources))

        self.connectPorts(initial.control_out, req.control_in)
        self.connectPorts(req.control_out, merge.control_in)
        self.connectPorts(choice.control_out[1], merge.control_in)
        self.connectPorts(merge.control_out, do_model.control_in)
        self.connectPorts(do_model.control_out, split.control_in)
        self.connectPorts(split.control_out, sim.control_in)
        self.connectPorts(split.control_out, check.control_in)
        self.connectPorts(sim.control_out, synchronization.control_in)
        self.connectPorts(check.control_out, synchronization.control_in)
        self.connectPorts(synchronization.control_out, evaluation.control_in)
        self.connectPorts(evaluation.control_out, choice.control_in)
        self.connectPorts(choice.control_out[0], self.finish.control_in)

        self.connectPorts(req.resource_out, resources.resource_in)
        self.connectPorts(resources.resource_out, req.resource_in)
        self.connectPorts(do_model.resource_out, resources.resource_in)
        self.connectPorts(resources.resource_out, do_model.resource_in)
        self.connectPorts(sim.resource_out, resources.resource_in)
        self.connectPorts(resources.resource_out, sim.resource_in)
        self.connectPorts(check.resource_out, resources.resource_in)
        self.connectPorts(resources.resource_out, check.resource_in)
        self.connectPorts(evaluation.resource_out, resources.resource_in)
        self.connectPorts(resources.resource_out, evaluation.resource_in)
```

Listing 21 Coupled DEVS model of the running example - Part 2.

```python
def simulate(nresources, verbose, seed=1):
    random.seed(seed)
    model = Example(nresources)
    sim = Simulator(model)

    sim.setClassicDEVS()
    sim.setTerminationCondition(lambda t, m: m.finish.state is not None)
    if verbose is not None:
        sim.setVerbose(verbose)
    sim.simulate()

    return model.finish.state

for i in range(1, 16):
    res = []
    for v in range(2,100):
        res.append(simulate(i, None, v))
    print(str(i) + " " + " ".join([str(i) for i in res]))
```

Appendix D

Artifacts of the proof of concept

This appendix documents the artifacts of the proof of concept.

Listing 22 The original design process encoded in Matlab. – Part 1

```
function ManualSolution( )
clc; close all;
%MANUALSOLUTION Show the results of a manual iteration process

addpath('ComponentSelection');
addpath('RSM_Toolbox');
addpath('MechanicalDesign');

MotorDB = MotorDBSetup();
BatteryDB = BatteryDBSetup();
MotorSelectionF = @(x) MotorSelection(x,MotorDB);
BatterySelectionF = @(x) BatterySelection(x,BatteryDB);

PdesV = 10;
CdesV = 1;

visualize1 = 1;
visualize2 = 1;
visualize3 = 1;
visualize4 = 1;
visualize5 = 1;
visualize6 = 1;
visualize7 = 1;
motorAnalytical = 1;
batteryAnalytical = 1;

%% -------------------------
% SETUP PROCESS DESCRIPTIONS
% --------------------------
%% 1. Design steps
designSteps = containers.Map();
    % Motor selection design step
motorSelection.parameters = {'Pdes';'Peff';'m';'r';'l'};
motorSelection.inputs = {'Pdes'};
motorSelection.outputs = {'m';'l';'r';'Peff'};
motorSelection.analytical = motorAnalytical;
motorSelection.functionCall = '[Peff,m,l,r] = MotorSelectionF(Pdes)';
motorSelection.data = containers.Map();
designSteps('motorSelection') = motorSelection;
    % Battery selection design step
batterySelection.parameters = {'Cdes';'Ceff';'m';'l';'w';'h'};
batterySelection.inputs = {'Cdes'};
batterySelection.outputs = {'m';'l';'w';'h';'Ceff'};
batterySelection.analytical = batteryAnalytical;
batterySelection.functionCall = '[Ceff,m,l,w,h] = BatterySelectionF(Cdes)';
batterySelection.data = containers.Map();
designSteps('batterySelection') = batterySelection;
    % Mechanical design design step
mechDesign.parameters = {'mBat';'lBat';'wBat';'hBat';'mMot';'lMot';'rMot';...
                         'mTot';'IxxTot';'IyyTot';'IzzTot'};
mechDesign.inputs = {'mBat';'lBat';'wBat';'hBat';'mMot';'lMot';'rMot'};
mechDesign.outputs = {'mTot';'IxxTot';'IyyTot';'IzzTot'};
mechDesign.analytical = 1;
mechDesign.functionCall = '[mTot, IxxTot, IyyTot, IzzTot] = MechanicalDesign(mBat,
    ↪ lBat,wBat,hBat,mMot,lMot,rMot)';
mechDesign.data = containers.Map();
designSteps('mechDesign') = mechDesign;
    % Controller optimization design step
contrOpt.parameters = {'mTot';'IxxTot';'IyyTot';'IzzTot';'PmaxMot';'TavgMot1';'
    ↪ TavgMot2';'Ku';'Tu'};
contrOpt.inputs = {'mTot';'IxxTot';'IyyTot';'IzzTot'};
contrOpt.outputs = {'PmaxMot';'TavgMot1';'TavgMot2'};
contrOpt.analytical = 0;
contrOpt.data = containers.Map();
designSteps('contrOpt') = contrOpt;
    % Electrical Simulation design step
elSim.parameters = {'TavgMot1';'TavgMot2';'CBat'};
elSim.inputs = {'TavgMot1';'TavgMot2'};
elSim.outputs = {'CBat'};
elSim.analytical = 0;
elSim.data = containers.Map();
designSteps('elSim') = elSim;
```

Listing 23 The original design process encoded in Matlab. – Part 2

```
%% 2. Connections map
sources = containers.Map();
sources('mechDesignPoint(''mBat'')')  = 'batterySelectionPoint(''m'')'; % [kg]
sources('mechDesignPoint(''lBat'')')  = 'batterySelectionPoint(''l'')'; % [m]
sources('mechDesignPoint(''wBat'')')  = 'batterySelectionPoint(''w'')'; % [m]
sources('mechDesignPoint(''hBat'')')  = 'batterySelectionPoint(''h'')'; % [m]
sources('mechDesignPoint(''mMot'')')  = 'motorSelectionPoint(''m'')';     % [kg]
sources('mechDesignPoint(''lMot'')')  = 'motorSelectionPoint(''l'')';     % [m]
sources('mechDesignPoint(''rMot'')')  = 'motorSelectionPoint(''r'')';     % [m]
sources('contrOptPoint(''mTot'')')      = 'mechDesignPoint(''mTot'')';      % [kg]
sources('contrOptPoint(''IxxTot'')')  = 'mechDesignPoint(''IxxTot'')';  % [kg*m^2]
sources('contrOptPoint(''IyyTot'')')  = 'mechDesignPoint(''IyyTot'')';  % [kg*m^2]
sources('contrOptPoint(''IzzTot'')')  = 'mechDesignPoint(''IzzTot'')';  % [kg*m^2]
sources('elSimPoint(''TavgMot1'')')     = 'contrOptPoint(''TavgMot1'')';  % [Nm]
sources('elSimPoint(''TavgMot2'')')     = 'contrOptPoint(''TavgMot2'')';  % [Nm]
assignInputs = @(x) assignInputsF(x,designSteps(x).inputs,sources);

%% Manual performance of iterations 1 and 2
motorSelectionPoint = containers.Map();
batterySelectionPoint = containers.Map();
mechDesignPoint = containers.Map();
contrOptPoint = containers.Map();
elSimPoint = containers.Map();
```

Listing 24 The original design process encoded in Matlab. – Part 3

```matlab
    % ITERATION 1
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)];
add(motorSelection.data,motorSelectionPoint);

batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,batterySelectionPoint)];
add(batterySelection.data,batterySelectionPoint);

assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
add(mechDesign.data,mechDesignPoint);

assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 94.86;   % Manually executed [kW]
contrOptPoint('TavgMot1') = 6.1266; % Manually executed [Nm]
contrOptPoint('TavgMot2') = 3.2249; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.6952; % Manually executed [1]
contrOptPoint('Tu')         = 0.1274; % Manually executed [1]
add(contrOpt.data,contrOptPoint);

assignInputs('elSim');
elSimPoint('CBat')  = 4.56; % [Ah]
add(elSim.data,elSimPoint);

% Generate RSMs
motorSelection.RSM = generateRSM(motorSelection);
batterySelection.RSM = generateRSM(batterySelection);
mechDesign.RSM = generateRSM(mechDesign);
contrOpt.RSM = generateRSM(contrOpt);
elSim.RSM = generateRSM(elSim);

[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,BatteryDB);

%% Visualize
if visualize1
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    fig = gcf;
    ResizeAndSave( fig,'Column1_Fig1' );
    saveas(fig,'Column2_Fig1.png');
    savefig(fig,'Column2_Fig1.fig');
    saveas(fig,'Column3_Fig1.png');
    savefig(fig,'Column3_Fig1.fig');
end
```

Listing 25 The original design process encoded in Matlab. – Part 4

```
    % ITERATION 2
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
add(motorSelection.data,motorSelectionPoint);

batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
add(batterySelection.data,batterySelectionPoint);

assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
add(mechDesign.data,mechDesignPoint);

assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 110.3;  % Manually executed [kW]
contrOptPoint('TavgMot1') = 10.3584;  % Manually executed [Nm]
contrOptPoint('TavgMot2') = 8.7828; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.6952; % Manually executed [1]
contrOptPoint('Tu')         = 0.1624; % Manually executed [1]
add(contrOpt.data,contrOptPoint);

assignInputs('elSim');
elSimPoint('CBat')  = 5.73108; % [Ah]
add(elSim.data,elSimPoint);

% Generate RSMs
motorSelection.RSM = generateRSM(motorSelection);
batterySelection.RSM = generateRSM(batterySelection);
mechDesign.RSM = generateRSM(mechDesign);
contrOpt.RSM = generateRSM(contrOpt);
elSim.RSM = generateRSM(elSim);

[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);

%% Visualize
if visualize2
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    ResizeAndSave( gcf,'Column1_Fig2' );
end
```

Listing 26 The original design process encoded in Matlab. – Part 5

```
    % ITERATION 3
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
add(motorSelection.data,motorSelectionPoint);

batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
add(batterySelection.data,batterySelectionPoint);

assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
add(mechDesign.data,mechDesignPoint);

assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 92.0; % Manually executed [kW]
contrOptPoint('TavgMot1') = 10.2638;  % Manually executed [Nm]
contrOptPoint('TavgMot2') = 8.8413; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.6190; % Manually executed [1]
contrOptPoint('Tu')         = 0.1468; % Manually executed [1]
add(contrOpt.data,contrOptPoint);

assignInputs('elSim');
elSimPoint('CBat')  = 13.7354; % [Ah]
add(elSim.data,elSimPoint);

% Generate RSMs
motorSelection.RSM = generateRSM(motorSelection);
batterySelection.RSM = generateRSM(batterySelection);
mechDesign.RSM = generateRSM(mechDesign);
contrOpt.RSM = generateRSM(contrOpt);
elSim.RSM = generateRSM(elSim);

[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);

%% Visualize
if visualize3
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    ResizeAndSave( gcf,'Column1_Fig3' );
end
```

Listing 27 The original design process encoded in Matlab. – Part 6

```
%% ITERATION 4
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
add(motorSelection.data,motorSelectionPoint);

batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
add(batterySelection.data,batterySelectionPoint);

assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
add(mechDesign.data,mechDesignPoint);

assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 98.4; % Manually executed [kW]
contrOptPoint('TavgMot1') = 8.9791; % Manually executed [Nm]
contrOptPoint('TavgMot2') = 7.7561; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.5109; % Manually executed [1]
contrOptPoint('Tu')         = 0.2154; % Manually executed [1]
add(contrOpt.data,contrOptPoint);

assignInputs('elSim');
elSimPoint('CBat')  = 3.33627; % [Ah]
add(elSim.data,elSimPoint);

% Generate RSMs
motorSelection.RSM = generateRSM(motorSelection);
batterySelection.RSM = generateRSM(batterySelection);
mechDesign.RSM = generateRSM(mechDesign);
contrOpt.RSM = generateRSM(contrOpt);
elSim.RSM = generateRSM(elSim);

[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);

%% Visualize
if visualize4
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    ResizeAndSave( gcf,'Column1_Fig4' );
end
```

Listing 28 The original design process encoded in Matlab. – Part 7

```
%% ITERATION 5
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 110.3;  % Manually executed [kW]
contrOptPoint('TavgMot1') = 10.3584;  % Manually executed [Nm]
contrOptPoint('TavgMot2') = 8.7828; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.6952; % Manually executed [1]
contrOptPoint('Tu')         = 0.1624; % Manually executed [1]
assignInputs('elSim');
elSimPoint('CBat')  = 5.73108; % [Ah]
[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);
if visualize5
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    ResizeAndSave( gcf,'Column1_Fig5' );
end
```

Listing 29 The original design process encoded in Matlab. – Part 8

```
%% ITERATION 6
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 92.0; % Manually executed [kW]
contrOptPoint('TavgMot1') = 10.2638;  % Manually executed [Nm]
contrOptPoint('TavgMot2') = 8.8413; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.6190; % Manually executed [1]
contrOptPoint('Tu')         = 0.1468; % Manually executed [1]
assignInputs('elSim');
elSimPoint('CBat')  = 13.7354; % [Ah]
[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);
if visualize6
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    ResizeAndSave( gcf,'Column1_Fig6' );
end
```

Listing 30 The original design process encoded in Matlab. – Part 9

```
%% ITERATION 7
motorSelectionPoint('Pdes') = PdesV(end); % [kW]
motorSelectionPoint = [motorSelectionPoint; call(motorSelection,motorSelectionPoint)
    ↪ ];
batterySelectionPoint('Cdes') = CdesV(end); % [Ah]
batterySelectionPoint = [batterySelectionPoint; call(batterySelection,
    ↪ batterySelectionPoint)];
assignInputs('mechDesign');
mechDesignPoint = [mechDesignPoint; call(mechDesign,mechDesignPoint)];
assignInputs('contrOpt');
contrOptPoint('PmaxMot')  = 98.4; % Manually executed [kW]
contrOptPoint('TavgMot1') = 8.9791; % Manually executed [Nm]
contrOptPoint('TavgMot2') = 7.7561; % Manually executed [Nm]
contrOptPoint('Ku')         = 0.5109; % Manually executed [1]
contrOptPoint('Tu')         = 0.2154; % Manually executed [1]
assignInputs('elSim');
elSimPoint('CBat')  = 3.33627; % [Ah]
[PdesV(end+1),CdesV(end+1)] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
    ↪ BatteryDB);
if visualize7
    visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,batterySelection,
        ↪ contrOptPoint('PmaxMot'),elSimPoint('CBat'));
    ResizeAndSave( gcf,'Column1_Fig7' );
end
```

Listing 31 The original design process encoded in Matlab. – Part 10

```
disp('Not done')

    % Assign all inputs
    function assignInputsF(designstep,properties,sources)
        for i = 1:length(properties)
            curProp = properties{i};
            fullName = [designstep 'Point(''' curProp ''')'];
            source = sources(fullName);
            evalin('caller',[fullName ' = ' source ';']);
        end
    end
    % Add a datapoint to the data set
    % The data set is a map, with parameter names as keys
    % The point is a struct, with parameter names as fields
    function add(data,point)
        pointParameters = keys(point);
        nParameters = length(pointParameters);
        if data.size(1)==0
            % If the data set is empty, keys are created based in point fields
            for i = 1:nParameters
                curKey = pointParameters{i};
                newData = point(curKey);
                data(curKey) = newData(:);
            end
        else
            dataParameters = keys(data);
            nDataPoints = length(data(dataParameters{1}));
            % If the data is not empty, test that its parameters are the
            % same as those of the point
            if ~isempty(setdiff(dataParameters,pointParameters))
                str = 'Data set uses different parameters as the provided point\n';
                str = [str 'Data parameters: ' dataParameters '\n'];
                str = [str 'Point parameters: ' pointParameters '\n'];
                error(str);
            end
            % Check if the distance to the previous points is not too large
            dataM = zeros(nDataPoints,nParameters);
            dataV = zeros(1,nParameters);
            for ii = 1:nParameters
                dataM(:,ii) = data(dataParameters{ii});
                dataV(:,ii) = point(dataParameters{ii});
            end
            dist = min(sqrt(sum(abs(dataM-ones(nDataPoints,1)*dataV).^2,2)));
            if  dist < 1E-6
              warning('New data point is too close to an existing point. Ignoring new
                  ↪  point')
              return;
            end
            % Add the data from the point
            for i = 1:nParameters
                curKey = pointParameters{i};
                newData = point(curKey);
                data(curKey) = [data(curKey); newData(:)];
            end
        end
    end
```

Listing 32 The original design process encoded in Matlab. – Part 11

```matlab
    % Generate a RSM for the given model with data
    function rsm = generateRSM(model)
        %% Extract data
        nInputs     = length(model.inputs);
        nOutputs    = length(model.outputs);
        inputs      = model.inputs;
        outputs     = model.outputs;
        nDataPoints = length(model.data(inputs{1}));
        dataM = zeros(nDataPoints,nInputs);
        for ii = 1:nInputs
            dataM(:,ii) = model.data(inputs{ii});
        end
        % Determine fitting function
        if nDataPoints >= nInputs+1
            fittingF = 'regpoly1';
        else
            fittingF = 'regpoly0';
        end
        % Setup the output struct
        % Add which model information was used that may change over time
        rsm.inputs      = inputs; % Inputs - Outputs may change
        rsm.outputs     = outputs;
        rsm.nDataPoints = nDataPoints; % Data may be added
        % Fit parameters
        rsm.dmodel  = containers.Map(); % Store a krigin model struct per output
        for ii = 1:nOutputs
            % RSM toolbox: fit Krigin RSM
            rsm.dmodel(outputs{ii}) = dacefit(dataM,model.data(outputs{ii}), fittingF
                ↪ , 'correxp', ones(1,nInputs), 1e-1*ones(1,nInputs), 20*ones(1,
                ↪ nInputs));
        end
    end
    % Evaluate a model at a given point
    % In case of an analytical design step, it calls the provided function call
    % In case of a non-analytical design step, evaluates its RSM
    function result = evaluateModel(model,point)
        if model.analytical
            result = call(model,point);
        else
            result = evaluateRSM(model,point);
        end
    end
```

Listing 33 The original design process encoded in Matlab. – Part 12

```matlab
% Evaluate a model using its given function call
function result = call(model,point)
    result = containers.Map();
    % Extract data
    nInputs = size(model.inputs,1);
    nOutputs= size(model.outputs,1);
    inputs  = model.inputs;
    outputs = model.outputs;
    functionCall = strrep(model.functionCall,' ','');
    % Split the function call into inputs, function name and outputs
    parts       = strsplit(functionCall,'=');
    outputSs     = strsplit(parts{1}(2:end-1),',');
    parts       = strsplit(parts{2},'(');
    functionS = parts{1};
    inputSs = strsplit(parts{2}(1:end-1),',');
    % Rebuild the function call
    functionCallParts = cell(0,1);
    functionCallParts{end+1} = '[';
    for ii = 1:length(outputSs)
        functionCallParts{end+1} = outputSs{ii};
        if ii ~= length(outputSs)
            functionCallParts{end+1} = ',';
        end
    end
    functionCallParts{end+1} = ['] = ' functionS '('];
    for ii = 1:length(inputSs)
        functionCallParts{end+1} = inputSs{ii};
        if ii ~= length(inputSs)
            functionCallParts{end+1} = ',';
        end
    end
    functionCallParts{end+1} = ');';
    % Rewrite all inputs and outputs to use the provided maps
    for ii = 1:nInputs
        for jj = 1:length(functionCallParts)
            if strcmp(inputs{ii},functionCallParts{jj})
                functionCallParts{jj} = ['point(''' inputs{ii} ''')'];
                break;
            end
        end
    end
    for ii = 1:nOutputs
        for jj = 1:length(functionCallParts)
            if strcmp(outputs{ii},functionCallParts{jj})
                functionCallParts{jj} = ['result(''' outputs{ii} ''')'];
                break;
            end
        end
    end
    functionCall = strjoin(functionCallParts,'');
    % Evaluate the expression
    eval([functionCall ';']);
end
```

Listing 34 The original design process encoded in Matlab. – Part 13

```
% Generate feasibility plot for the given RSMs of the design process
function visualizeNextStep(PdesV,CdesV,MotorDB,BatteryDB,motorSelection,
    ↪ batterySelection,Plast,Clast)
        % POWERS
    pList = [MotorDB.P];
    pStep = mean(pList(2:end)-pList(1:end-1));
    pMin = min(pList)-3*pStep;
    pMax = max(pList);
    pV = pMin:pStep/2:pMax;
        % CAPACITIES
    cList = [BatteryDB.C];
    cStep = mean(cList(2:end)-cList(1:end-1));
    cMin = min(cList)-3*cStep;
    cMax = max(cList);
    cV = cMin:cStep/2:cMax; %linspace(cMin,cMax,100);
    feas = zeros(length(pV),length(cV));
    % Loop over all points
    for i=1:length(pV)
        for j = 1:length(cV)
            feas(i,j) = Plast<pV(i) && Clast<cV(j);
        end
    end
    % Plot
    plotNextStep(PdesV(end-1:end),CdesV(end-1:end),MotorDB,BatteryDB,
        ↪ motorSelection,batterySelection,cV,pV,feas);
end
```

Listing 35 The original design process encoded in Matlab. – Part 14

```
% Generate feasibility plot for the given RSMs of the design process
function [curOptP,curOptC] = findNextPoint(contrOptPoint,elSimPoint,MotorDB,
     ↪ BatteryDB)
    % First motor & battery that can provide this
    curOptP = MotorDB(find([MotorDB.P]>=contrOptPoint('PmaxMot'),1,'first')).P;
    curOptC = BatteryDB(find([BatteryDB.C]>=elSimPoint('CBat'),1,'first')).C;
end
% Generate feasibility plot for the given RSMs of the design process
function [c,ceq] = evaluateConstraints(Pdes,Cdes,motorSelection,...
    batterySelection,mechDesign,contrOpt,elSim)

    % Initialize all input points
    motorSelectionInput = containers.Map();
    batterySelectionInput = containers.Map();
    mechDesignInput = containers.Map();
    contrOptInput = containers.Map();
    elSimInput = containers.Map();

    % Initial motor and battery dimensioning
    motorSelectionInput('Pdes') = Pdes; % [kW]
    batterySelectionInput('Cdes') = Cdes; % [Ah]

    % Evaluate motor and battery selection models
    motorSelectionOutput = evaluateModel(motorSelection, motorSelectionInput);
    batterySelectionOutput = evaluateModel(batterySelection,
         ↪ batterySelectionInput);
    Peff = motorSelectionOutput('Peff');
    Ceff = batterySelectionOutput('Ceff');

    % Insert into mechanical design model input
    mechDesignInput('mBat') = batterySelectionOutput('m');
    mechDesignInput('lBat') = batterySelectionOutput('l');
    mechDesignInput('wBat') = batterySelectionOutput('w');
    mechDesignInput('hBat') = batterySelectionOutput('h');
    mechDesignInput('mMot') =   motorSelectionOutput('m');
    mechDesignInput('lMot') =   motorSelectionOutput('l');
    mechDesignInput('rMot') =   motorSelectionOutput('r');

    % Evaluate mechanical design model
    mechDesignOutput = evaluateModel(mechDesign, mechDesignInput);

    % Insert into controller optimization input
    contrOptInput(  'mTot') = mechDesignOutput(  'mTot');
    contrOptInput('IxxTot') = mechDesignOutput('IxxTot');
    contrOptInput('IyyTot') = mechDesignOutput('IyyTot');
    contrOptInput('IzzTot') = mechDesignOutput('IzzTot');

    % Evaluate controller optimization model
    contrOptOutput = evaluateModel(contrOpt, contrOptInput);

    % Insert into controller optimization input
    elSimInput('TavgMot1') = contrOptOutput('TavgMot1');
    elSimInput('TavgMot2') = contrOptOutput('TavgMot2');

    % Evaluate controller optimization model
    elSimOutput = evaluateModel(elSim, elSimInput);

    % Extract required values
    Preq = contrOptOutput('PmaxMot');
    Creq = elSimOutput('CBat');

    % Feasibility
    ceq = [];
    c = [Preq-Peff; Creq-Ceff];
end
```

Listing 36 The original design process encoded in Matlab. – Part 15

```
    % Visualize a single input-output relationship
    % Inputs:
    % - RSM model
    % - Point containing the value of all other inputs
    % - Name of the input to vary
    % - Name of the output to get
    function visualizeInputOutputRelation(model,point,input,output,fig,opt)
        % Create range
        inputData = model.data(input);
        inputMin = min(inputData)*0.9;
        inputMax = max(inputData)*1.1;
        inputVals = linspace(inputMin,inputMax,100);
        outputVals = zeros(size(inputVals));
        for ii = 1:length(inputVals)
            inputVal = inputVals(ii);
            point(input) = inputVal;
            result = evaluateModel(model,point);
            outputVals(ii) = result(output);
        end
        figure(fig);
        hold on;
        plot(inputVals,outputVals,opt)
        xlabel(input)
        ylabel(output)
    end
end
```

Listing 37 The motor database setup activity of the demonstrative example.

```
function MotorSelection = MotorSelection()
MotorSelection = [];
%% Settings
powerToWeight = 2; % [kW/kg] --> based on UQM PP 100
density = 50/(0.233*(0.286/2)^2*pi); % [kg/m3] --> based on UQM PP 100

%% Interpolate to create a "limited" database
Pvect = 10:10:200; % [kW]
for i = 1:length(Pvect)
    Pcur = Pvect(i);
    m = Pcur/powerToWeight;
    r = 0.286/2;
    l = m/(density*pi*r^2);
    MotorSelection = addOne(MotorSelection,Pcur,m,l,r);
end

%% Output
MotorSelection = orderByField(MotorSelection,'P');

    % Add a single battery
    function cur = addOne(cur,P,m,l,r)
        new.P = P;
        new.m = m;
        new.l = l;
        new.r = r;
        cur = [cur; new];
    end
    % Order according to the given field
    function res = orderByField(in,fieldName)
        fieldContent = [in.(fieldName)];
        [~,iOrdering] = sort(fieldContent);
        res = in(iOrdering);
    end

end
```

Listing 38 The battery database setup activity of the demonstrative example.

```
function BatterySelection = BatterySelection()
BatterySelection = [];
%% Load data
dataArray = csvread('KokamLargeBatteries.csv',1,0);
C.data = dataArray(:, 1); % [Ah]
w.data = dataArray(:, 2)./1000; % [m]
l.data = dataArray(:, 3)./1000; % [m]
t.data = dataArray(:, 4)./1000; % [m]
m.data = dataArray(:, 5); % [kg]

%% Create least-squares model
w.coef = [ones(size(C.data)) C.data]\w.data;
l.coef = [ones(size(C.data)) C.data]\l.data;
t.coef = [ones(size(C.data)) C.data]\t.data;
m.coef = [ones(size(C.data)) C.data]\m.data;

%% Interpolate to create a "limited" database
Cvect = 2:10:102;
for Ccur = Cvect
    BatterySelection = addOne(BatterySelection,Ccur,...
        [1 Ccur]*m.coef,...
        [1 Ccur]*w.coef,...
        [1 Ccur]*l.coef,...
        [1 Ccur]*t.coef);
end

%% Output
BatterySelection = orderByField(BatterySelection,'C');

    % Add a single battery
    function cur = addOne(cur,C,m,l,w,h)
        new.C = C;
        new.m = m;
        new.l = l;
        new.w = w;
        new.h = h;
        cur = [cur; new];
    end
    % Order according to the given field
    function res = orderByField(in,fieldName)
        fieldContent = [in.(fieldName)];
        [~,iOrdering] = sort(fieldContent);
        res = in(iOrdering);
    end

end
```

Listing 39 The original source of the battery DB.

```
 Capacity,w,l,t,m
 13,220,132,6.3,0.332
 12,220,132,6.5,0.34
 16,220,132,7.8,0.406
 25,226,227,6.3,0.6
 31,226,227,7.5,0.72
 40,226,227,9.3,0.9
 46,226,227,12.5,1.265
 53,226,227,12,1.16
 30,198,220,9.9,0.84
 52,268,265,9.3,1.26
 55,268,265,10.3,1.45
 63,268,265,11,1.52
 75,268,265,12,1.63
 87,268,265,13,1.78
 65,268,265,13.5,1.85
 75,268,265,13.3,2
 70,462,327,5.4,1.63
 80,462,327,6.3,1.92
 100,462,327,7,2.07
 150,462,327,10.5,3.21
 200,462,327,13.7,4.18
 240,462,327,15.8,4.78
```

Listing 40 The battery selection activity of the demonstrative example.

```
function [ Ceff,m,l,w,h ] = BatterySelection( Cdes,BatteryDB )
%BATTERYSELECTION

iBat   = find([BatteryDB.C]>=Cdes,1,'first');
Ceff   = BatteryDB(iBat).C; % [Ah]
m        = BatteryDB(iBat).m; % [kg]
l        = BatteryDB(iBat).l; % [m]
w        = BatteryDB(iBat).w; % [m]
h        = BatteryDB(iBat).h; % [m]

end
```

Listing 41 The motor selection activity of the demonstrative example.

```
function [ Peff,m,l,r ] = MotorSelection( Pdes,MotorDB )
%MOTORSELECTION

iMot   = find([MotorDB.P]>=Pdes,1,'first');
Peff   = MotorDB(iMot).P; % [kW]
m        = MotorDB(iMot).m; % [kg]
l        = MotorDB(iMot).l; % [m]
r        = MotorDB(iMot).r; % [m]

end
```

Listing 42 The mechanical design and simulation activity of the demonstrative example.

```matlab
function [mTot, IxxTot, IyyTot, IzzTot] = MechanicalDesign(mBat,lBat,wBat,hBat,...
    mMot,lMot,rMot)

% Assumptions:
% Battery is placed at the center of the robot
    % length = y-axis, width = x-axis, height = z-axis
Bat.x = 0;
Bat.y = 0;
Bat.z = hBat/2;
Bat.m = mBat;
elements = {Bat};
% Motors are placed at opposite ends of the battery
    % parallel to the y-axis, perpendicular to the x and z-axes
Mot1.x = 0;
Mot1.y = wBat/2+lMot/2;
Mot1.z = rMot;
Mot1.m = mMot;
Mot2.x = 0;
Mot2.y = -(wBat/2+lMot/2);
Mot2.z = rMot;
Mot2.m = mMot;
elements{end+1} = Mot1;
elements{end+1} = Mot2;
% Magnets are placed next to the batteries in the x-direction
Mag1.x = lBat/2+0.087/2;
Mag1.y = 0;
Mag1.z = 0;
Mag1.m = 2.989;
Mag2.x = -(lBat/2+0.087/2);
Mag2.y = 0;
Mag2.z = 0;
Mag2.m = 2.989;
elements{end+1} = Mag1;
elements{end+1} = Mag2;
% Base is a circle,
% with a plate thickness determined by the battery and motor mass
Base.x = 0;
Base.y = 0;
Base.z = 0;
Base.rho = 7800;
Base.r = wBat/2+lMot;

mOnBase = 0;
for curEl = elements
    mOnBase = mOnBase + curEl{1}.m;
end
Base.t = 3*mOnBase/1E5; % Linear approximation of the thickness

Base.m = Base.rho * pi * Base.r^2 * Base.t;
Base.Ixx = 1/12*Base.m*(3*Base.r^2+Base.t^2);
Base.Iyy = 1/12*Base.m*(3*Base.r^2+Base.t^2);
Base.Izz = 1/2*Base.m*Base.r^2;

elements{end+1} = Base;
% Sum all elements
mTot = 0; IxxTot = Base.Ixx; IyyTot = Base.Iyy; IzzTot = Base.Izz;
for curEl = elements
    mTot = mTot + curEl{1}.m;
    IxxTot = IxxTot + curEl{1}.m*(curEl{1}.y^2+curEl{1}.z^2);
    IyyTot = IyyTot + curEl{1}.m*(curEl{1}.x^2+curEl{1}.z^2);
    IzzTot = IzzTot + curEl{1}.m*(curEl{1}.x^2+curEl{1}.y^2);
end

end
```

Listing 43 The exported plant model from Virtual.Lab-Motion to Matlab/Simulink.

```
motionfiles =[
'C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1,
C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1,
C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1,
C:\agv\MobilePlatformToolDevelopment\Models\MechanicalModel\AnalysisCase.1.1'];
antype = ['cosim']; feedthrough = ['false'];
vl = getenv('VLMOTIONSLV');
vl1 = strcat(vl,'\execute\intel64');
path(path,vl1);
if ( strcmp(gcs,'') )
   disp('Error Matlab model not loaded');
   clear vl vl1;
   return;
end
subsys = strcat(gcs,'/plantout');
new = 0;
try
   add_block('built-in/SubSystem', subsys, 'Position', [200 200 300 300]  );
   new = 1;
catch
   try
      cnt = get_param( strcat(subsys,'/thedemux') , 'Outputs');
      for i = 1:str2num(cnt)
         delete_line( subsys, strcat('thedemux/',num2str(i)), strcat('themux/',
              ↪ num2str(i)) );
      end
      delete_line(subsys,'in/1', 'thebus/1' );
      delete_line(subsys,'thebus/1','motionfun/1' );
      delete_line(subsys,'themux/1','out/1');
      delete_line(subsys,'motionfun/1','thedemux/1');
      delete_block( strcat(subsys,'/motionfun') );
      delete_block( strcat(subsys,'/thedemux') );
      delete_block( strcat(subsys,'/themux') );
      delete_block( strcat(subsys,'/thebus') );
   catch
      warning 'Unable to modify plantout block will delete';
      delete_block( subsys );
      add_block('built-in/SubSystem', subsys, 'Position', [200 200 300 300]  );
      new = 1;
   end
end
if ( new )
   add_block('built-in/Inport', strcat( subsys, '/in'), 'Position', [15 125 65 175] )
         ↪ ;
   add_block('built-in/Outport', strcat( subsys, '/out'), 'Position', [735 125 785
         ↪ 175] );
end
add_block('built-in/Demux', strcat(subsys,'/thedemux'), 'Outputs', '2', 'Position',
      ↪ [365 100 465 200] );
add_block('built-in/BusCreator', strcat(subsys,'/themux'), 'Inputs', '2', 'Position',
      ↪  [580 100 590 200] );
add_block('built-in/BusSelector', strcat(subsys,'/thebus'), 'Position', [120 100 130
      ↪ 200], 'MuxedOutPut', 'On' );
sfunction_setup = 1;
add_block('built-in/S-Function', strcat(subsys,'/motionfun'), 'Position', [210 100
      ↪ 290 200], 'FunctionName', 'vlmotionmex', 'Parameters', 'antype,motionfiles,
      ↪ feedthrough' );
clear sfunction_setup;
add_line(subsys,'themux/1','out/1');
add_line(subsys,'in/1','thebus/1');
add_line(subsys,'thebus/1','motionfun/1');
add_line(subsys,'motionfun/1','thedemux/1');
a = add_line( subsys, 'thedemux/1', 'themux/1');
set_param( a, 'Name', 'a1' );
a = add_line( subsys, 'thedemux/2', 'themux/2');
set_param( a, 'Name', 'a2' );
set_param( strcat(subsys,'/thebus'), 'OutputSignals', 'T1,T2');
clear vl vl1 a i new subsys;
disp('setup for Virtual.Lab Motion done');
```

Listing 44 The querying template for AMESim.

```
class AmesimQuery extends AmesimAttributeDefinitionImpl implements IExecutable {

  private val logger = Logger.getLogger(this.class)
  private val AMESIM_LOCATION = new File("D:\\tools\\LMS\\LMS Imagine.Lab Amesim\\
      ↪ v1400")
  private val QUERY_RESULT_LABEL = "queryresult:"
  private val ERROR_MSG = "From AMEGetParameterValue: fail to locate parameter or
      ↪ variable in the active circuit:"

  private static extension val AliasHandler aliasHandler = new AliasHandler()

  @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Attribute attribute;
  @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Activity activity;

  new(Attribute attribute, Activity activity) {
    this.attribute = attribute
    this.activity = activity
  }

  override execute() {
    logger.level = Level::DEBUG

    val file = File.createTempFile("amesimquery", ".py", AMESIM_LOCATION)
    file.deleteOnExit()
    val bufferedWriter = new BufferedWriter(new FileWriter(file))

    val scriptText = AmesimQueryScriptTemplate.generateAmesimQueryScript(attribute.
        ↪ getQueryableName(activity)).toString
    bufferedWriter.write(scriptText)
    bufferedWriter.close();

    logger.debug(file.getAbsolutePath())
    logger.debug(scriptText)

    val command = '"' + AMESIM_LOCATION + '\\python.bat" "' + file.absolutePath + '"'
    logger.debug('command: ' + command)

    val processBuilder = new ProcessBuilder(command)
    processBuilder.directory(AMESIM_LOCATION)
    val process = processBuilder.start

    val bufferedReader = new BufferedReader(new InputStreamReader(process.inputStream
        ↪ ))
    val errorReader = new BufferedReader(new InputStreamReader(process.errorStream))

    evaluate(bufferedReader, errorReader)
  }

  private def evaluate(BufferedReader bufferedReader, BufferedReader errorReader) {
    val List<String> blines = Lists::newArrayList(bufferedReader.lines().toArray)
    if (!blines.empty) {
      for (line : blines) {
        logger.debug(line)
        if (line.trim.startsWith(QUERY_RESULT_LABEL)) {
          return Double.parseDouble(line.split(':').last.trim)
        }
      }
    }
  }
}
```

Listing 45 The querying template for Matlab.

```
class MatlabQuery extends MatlabAttributeDefinitionImpl implements IExecutable {

  private static extension val AliasHandler aliasHandler = new AliasHandler()

  @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Attribute attribute;
  @Accessors(PRIVATE_GETTER, PRIVATE_SETTER) private val Activity activity;

  new(Attribute attribute, Activity activity) {
    this.attribute = attribute
    this.activity = activity
  }

  override execute() {
    try {
      val matlabEngine = MatlabConnectionManager::matlabEngine;
      val queryableName = attribute.getQueryableName(activity)
      matlabEngine.getVariable(queryableName)
    } catch (Exception ex) {
      return null
    }
  }
}
```

Listing 46 The executor for Python.

```
class PythonExecutor {

  def execute(PythonScript script) {
    //val p = Runtime.getRuntime().exec("d:\\GitHub\\msdl\\ICM\\examples\\be.
         ↪ uantwerpen.msdl.icm.cases.demo\\python\\pythonstep.bat amesimtest.py")
    //this was added due to the standalone python installation packaged with LMS
         ↪ tools (e.g. AMESim)
    val p = Runtime.getRuntime().exec("python " + script.getScriptLocation);
    val in = new BufferedReader(new InputStreamReader(p.getInputStream()));
    System.out.println(in.readLine);
  }
}
```

Listing 47 The command executor for Matlab extending the *ParameterizedExecutor* in Listing 48.

```
enum ExecutionMode {
  NORMAL,
  HEADLESS
}

class MatlabExecutor extends ParameterizedExecutor {
  @Accessors(NONE) val DEFAULT_MODE = ExecutionMode::HEADLESS

  def execute(MatlabScript script, MatlabProxy matlabProxy, EMap<String, String>
      ↪ parameters,
    ExecutionMode executionMode) {
    switch (executionMode) {
      case NORMAL: executeWithGui(script, matlabProxy, parameters)
      case HEADLESS: executeHeadless(script, matlabProxy, parameters)
    }
  }

  def execute(MatlabScript script, MatlabProxy matlabProxy, EMap<String, String>
      ↪ parameters) {
    execute(script, matlabProxy, parameters, DEFAULT_MODE)
  }

  private def executeHeadless(MatlabScript script, MatlabProxy matlabProxy, EMap<
      ↪ String, String> parameters) {
    val path = Paths.get(script.scriptLocation)
    val charset = StandardCharsets.UTF_8
    val rawContent = new String(Files.readAllBytes(path), charset)

    // resolve parameters in the next line of the script
    val content = rawContent.resolveParameters(parameters)

    // execute command
    val matlabEngine = MatlabConnectionManager::matlabEngine
    matlabEngine.eval(content)
  }

  private def executeWithGui(MatlabScript script, MatlabProxy matlabProxy, EMap<
      ↪ String, String> parameters) {
    val file = new File(script.scriptLocation)

    try {
      val bufferedReader = new BufferedReader(new FileReader(file))
      var String line = ""

      while ((line = bufferedReader.readLine()) !== null) {
        // resolve parameters in the next line of the script
        line = line.resolveParameters(parameters)

        // execute command
        matlabProxy.eval(line)

        // update variable store
        line.extractAssignments
      }
    } catch (Exception e) {
      e.printStackTrace
    }
  }

}
```

Listing 48 The parameterized command executor. Serves as a superclass for platform-specific executors.

```
class ParameterizedExecutor {

  val BYNUMBER_NAME_PATTERN = "\\%\\{args\\[i\\].name\\}\\%"
  val BYNUMBER_VALUE_PATTERN = "\\%\\{args\\[i\\].value\\}\\%"
  val BYNAME_NAME_PATTERN = "\\%\\{args\\['n'\\].name\\}\\%"
  val BYNAME_VALUE_PATTERN = "\\%\\{args\\['n'\\].value\\}\\%"

  val ASSIGNMENT_PATTERN = "varname\\s*=\\s*[0-9]*"

  def resolveParameters(String command, EMap<String, String> parameters) {
    var String newCommand = command

    for (parameter : parameters) {
      val key = parameter.key
      val value = parameter.value
      val index = parameters.indexOfKey(key)

      newCommand = newCommand.replaceAll(BYNUMBER_NAME_PATTERN.replace('i', index.
          ↪ toString), key)
      newCommand = newCommand.replaceAll(BYNUMBER_VALUE_PATTERN.replace('i', index.
          ↪ toString), value)
      newCommand = newCommand.replaceAll(BYNAME_NAME_PATTERN.replaceFirst('n', key),
          ↪ key)
      newCommand = newCommand.replaceAll(BYNAME_VALUE_PATTERN.replaceFirst('n', key),
          ↪  value)
    }

    newCommand = newCommand.replaceUnused

    newCommand
  }

  def getReplaceUnused(String command) {
    var String newCommand = command
    newCommand = newCommand.replaceAll("\\%\\{args\\['[a-zA-Z0-9_]*'\\].value\\}\\%",
        ↪  'null')
    newCommand
  }

  def extractAssignments(String line) {
    val variableManager = VariableManager.instance

    for (variable : variableManager.variableStore.variables) {
      val pattern = ASSIGNMENT_PATTERN.replace('varname', variable.name)
      val matches = Pattern.compile(pattern).matcher(line)
      while (matches.find) {
        val split = matches.group.split("=").toList
        val varname = split.head
        val value = new Double(split.last)

        variableManager.setVariable(varname, value)
      }
    }
  }
}
```

Appendix E

Formal notations

This appendix lists the notations used in the formal parts of this work.

- $a \in A$ – activity, set of activities
- $b(\omega)$ – availability of the resource ω
- B – vector of availabilities
- c_e – cost per execution
- c_p – cost for presence
- c_u – cost per unit
- $C(\pi)$ – cost of process π
- $\gamma \in \Gamma$ – capability, set of capabilities
- $d \in D$ – design artifact, set of design artifacts
- $\delta_c \in \Delta_c : A \mapsto A$ – control flow
- $\delta_d \in \Delta_d : A \mapsto D$ – design artifact flow
- Δ_c^+ – transitive closure of an activity
- $\bigcup_{d_i \subseteq D} d_i$ – virtual product
- $f \in \mathcal{F}$ – formalism, set of formalisms
- $i \in I$ – intent, set of intents
- $k(a)$ – resource demand of activity a
- $\chi(\xi)$ – change scope of the system characteristic ξ
- $\chi^+(\xi_i)$ – the transitive closure of the change scope over system characteristic ξ_i
- $l(a)$ – resource allocation of activity a
- Λ – levels of precision

- $m \in M$ – management pattern, set of management patterns
- μ – marking of process
- $\omega \in \Omega$ – resource, resources
- $p \in P$ – property, set of properties
- $P_{req}(D_i) \subseteq P$ – the set of properties required to be satisfied by D_i
- $P_{sat}(D_i) \subseteq P$ – the set of properties satisfied by D_i
- $P_{unsat}(D_i) \subseteq P$ – the set of required but not satisfied properties by D_i
- par(a_1, a_2) – parallel activities
- $\pi \in \Pi$ – process, set of processes
- Ψ – set of inconsistencies
- $\psi(\xi)$ – inconsistency mapping function of system characteristic ξ
- $r \in R$ – relationship, set of relationships
- $\mathcal{S}$ – system
- seq(a_1, a_2) – sequential activities
- Σ_Π – process space
- $t \in \mathcal{T}$ – transformation, set of transformations
- $t_I \in T_I$ – type of intent, set of all types of intents
- $\theta \in \Theta$ – attribute, set of attributes
- $\xi \in \Xi$ – system characteristic, set of system characteristics

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] ACM. ACM Digital Library. `https://dl.acm.org`. Acc: 2019-05-11. 23

[5] 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. 31, 33, 38, 39, 40

[6] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. 18, 170

[7] 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

[8] 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. 47

[9] A. I. Antón, W. M. McCracken, and C. Potts. Goal decomposition and scenario analysis in business process reengineering. In *International Conference on Advanced Information Systems Engineering*, pages 94–104. Springer, 1994. 10

[10] Apache Software Foundation. Apache Maven Website. `https://maven.apache.org`. Acc: 2017-08-17. 19

[11] C. Artigues, S. Demassey, and E. Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2007. 47, 72

[12] 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

[13] 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

[14] R. Balzer. Tolerating inconsistency. *[1991 Proceedings] 13th International Conference on Software Engineering*, pages 158–165, 1991. 31, 39, 41, 167, 170

[15] 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

[16] 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. 170

[17] S. Bechhofer. OWL: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009. 71

[18] 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. 31, 33, 38, 39

[19] 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, 56

[20] 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

[21] 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. 44

[22] K. Berx, H. Karhula, M. Nicolai, and M. Davy. Extended dependency modelling for effective and efficient design decision support. (Technical report). 61

[23] J. Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004. 12

[24] 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

[25] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. Multi-domain modeling of cyber-physical systems using architectural views. *AVICPS 2010*, page 43, 2010. 31, 34, 39

[26] 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. 31, 35, 39

[27] 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

[28] G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22–27, 2009. 160, 165

[29] 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. 31, 35, 38, 39, 41

[30] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988. 18

[31] N. Bouillot and E. Gressier-Soudan. Consistency models for distributed interactive multimedia applications. *SIGOPS Operating Systems Review*, 38(4):20–32, Oct. 2004. 44

[32] G. E. Box. Robustness in the strategy of scientific model building. In *Robustness in statistics*, pages 201–236. Elsevier, 1979. 12

[33] 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. 16, 56

[34] T. R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering management*, 48(3):292–306, 2001. 48

[35] B. Choudhary. *The Elements of Complex Analysis*. New Age International, 1992. ISBN 978-81-224-0399-2. 106

[36] 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

[37] CMMI Institute. CMMI. `http://cmmiinstitute.com/capability-maturity-model-integration`. Accessed: 2018-11-20. 9, 91, 106, 168

[38] 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

[39] M. Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004. 10

[40] 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. XVII, 16

[41] K. Craig. Multidisciplinary Mechatronic Innovations. `http://www.multimechatronics.com`. Acc: 2019-05-12. 2

[42] M. Cumberlidge. *Business process management with JBoss jBPM*. Packt Publishing Ltd, 2007. 45

[43] 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. 31, 34, 39, 40

[44] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1-2):3–50, 1993. 10

[45] 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, 77, 101, 108, 120, 129

[46] 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, 120

[47] 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

[48] I. Dávid, I. Ráth, and D. Varró. Foundations for Streaming Model Transformations by Complex Event Processing. *Software and Systems Modeling*. 40

[49] 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. 144, 170

[50] 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, pages 616–627. IEEE, Dec. 2018. 106

[51] K. Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014. 145

[52] J. Denil. Design, verification and deployment of software-intensive systems: a multi-paradigm modelling approach. *University of Antwerp*, 2013. 92

[53] J. Denil, R. Salay, C. Paredis, and H. Vangheluwe. Towards agile model-based systems engineering. In *CEUR workshop proceedings*, pages 424–429, 2017. 9

[54] 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

[55] 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

[56] 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. 31, 39, 41

[57] Eclipse Foundation. Eclipse Modeling Framework (EMF) Website. `https://eclipse.org/modeling/emf/`. Acc: 2017-08-17. 91, 152

[58] Eclipse Foundation. EMFCompare. `https://www.eclipse.org/emf/compare/`. Accessed: 2015-11-20. 38

[59] 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

[60] Eclipse Foundation. Sirius Website. `https://eclipse.org/sirius/`. Acc: 2017-07-07. 65, 98, 153

[61] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, March 2011. 31, 33, 38, 39, 104

[62] 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. 31, 34, 38, 39

[63] 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. 31, 36, 39

[64] 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

[65] S. D. Eppinger and T. R. Browning. *Design structure matrix methods and applications*. MIT press, 2012. 47, 154

[66] R. Eramo, A. Pierantonio, and G. Rosa. Approaching Collaborative Modeling as an Uncertainty Reduction Process. In *COMMitMDE@ MoDELS*, pages 27–34, 2016. 104

[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. 170

[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. 2, 27, 30, 41, 51

[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. 2, 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] 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

[76] 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. 31, 33, 39, 40

[77] D. A. Gebala and S. D. Eppinger. Methods for analyzing design procedures. 1991. 48

[78] 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. 31, 37, 39

[79] 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. 31, 33, 38, 39, 40

[80] 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

[81] D. Gross, J. Shortle, J. Thompson, and C. Harris. *Fundamentals of Queueing Theory*. Wiley, 2011. 88

[82] 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. 2, 17

[83] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007. 27

[84] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. 19, 46, 120, 121

[85] 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, 43

[86] Á. Hegedüs, Á. Horváth, and D. Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering*, 22(3):399–436, 2015. 81, 86

[87] 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. 26, 31, 36, 38, 39

[88] 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. 31, 37, 39

[89] 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. 26

[90] 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. 2

[91] 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. 31, 32, 38, 39, 40, 43

[92] 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

[93] A. Hunter, S. Konieczny, et al. Measuring inconsistency through minimal inconsistent sets. *KR*, 8:358–366, 2008. 42

[94] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd international conference on software engineering*, pages 471–480. ACM, 2011. 137

[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] IEEE. IEEE Xplore Digital Library. `https://ieeexplore.ieee.org/Xplore/home.jsp`. Acc: 2019-05-11. 23

[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`. 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`. 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. 44

[111] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979. 44

[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. 31, 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. 123

[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] M. Li and D. Li. Modular decomposition method based on design structure matrix and application. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 10(8):2173–2175, 2012. 48

[118] 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

[119] 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. 31, 39

[120] 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. 31, 37, 39

[121] 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. 29

[122] 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. 19, 63, 109, 110

[123] 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

[124] MathWorks Inc. MATLAB API for Java. https://www.mathworks.com/help/matlab/matlab-engine-api-for-java.html. Acc.: 2016-08-02. 162

[125] MathWorks Inc. Matlab Website. mathworks.com/products/matlab. Acc.: 2016-08-02. 152

[126] MathWorks Inc. Multibody Modeling. https://www.mathworks.com/help/physmod/sm/multibody-modeling.html. Acc.: 2018-12-16. 140

[127] MathWorks Inc. SimEvents Website. mathworks.com/products/simevents. Acc.: 2016-08-02. 88

[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. 31, 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. 157

[130] Microsoft. C# Reference. `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/`. Accessed: 2019-05-04. 14

[131] M. Minsky. Matter, mind and models. 1965. 11

[132] 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

[133] modelpractice. General Model Theory by Stachowiak. `https://modelpractice.wordpress.com/2012/07/04/model-stachowiak/`. 11

[134] P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004. 1

[135] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 19

[136] S. Mustafiz and H. Vangheluwe. Explicit modelling of statechart simulation environments. Proceedings of the Summer Simulation Multiconference, pages 21:1–21:8, 2013. 129

[137] 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. 31, 39, 40

[138] I. Nonaka. *The knowledge-creating company*. Harvard Business Review Press, 2008. 137

[139] OASIS. Human Task (WS-HumanTask) Version 1.1. `http://docs.oasis-open.org/ns/bpel4people/ws-humantask/200803`. Acc: 2017-08-17. 19

[140] 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

[141] Object Management Group (OMG). SysML Website. `http://www.omgsysml.org/`. Accessed: 2018-11-20. 16

[142] Object Management Group (OMG). UML Website. `https://www.omg.org/spec/UML`. Accessed: 2019-05-04. 14

[143] Object Management Group (OMG). BPMN 2.0 Specification. `http://www.bpmn.org/`, 2011. Accessed: 2018-05-06. 19

[144] 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. 31, 32, 39

[145] OSLC Community. OSLC - Open services for lifecycle collaboration core specification version 3.0. http://open-services.net, 2017. Accessed: 2018-11-20. 45

[146] 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, 120

[147] 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

[148] Oxford Dictionaries. Definition of "process". https://en.oxforddictionaries.com/definition/process. Accessed: 2018-11-20. 17

[149] G. Pahl and W. Beitz. *Engineering design: a systematic approach*. Springer Science & Business Media, 2013. 18

[150] 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. 129

[151] M. Persson, M. Törngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil. A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In *Proceedings of the International Conference on Embedded Software*, 2013. 1, 2, 57, 59

[152] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *Ease*, volume 8, pages 68–77, 2008. 21, 22

[153] 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

[154] 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. 31, 33, 38, 39, 40

[155] X. Qin. Delayed consistency model for distributed interactive systems with real-time continuous media. *Journal of Software*, 6(13):1029–1039, 2002. 44

[156] 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. 31, 36, 38, 39

[157] 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. 148

[158] 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. 170

[159] 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

[160] 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. 26

[161] M. H. Romanycia and F. J. Pelletier. What is a heuristic? *Computational Intelligence*, 1(1):47–58, 1985. 3, 51

[162] 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. 170

[163] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, volume 40, pages 167–176. ACM, 2005. 48

[164] 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

[165] 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

[166] B. Schätz, F. Hölzl, 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

[167] A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995. 32

[168] Scrum Alliance. Learn About Scrum. `https://www.scrumalliance.org/learn-about-scrum`. Accessed: 2018-11-20. 9

[169] B. Selic. A short catalogue of abstraction patterns for model-based software engineering. *Int. J. Software and Informatics*, 5(1-2):313–334, 2011. 59

[170] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003. 125, 168

[171] 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. 44

[172] 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. 31, 34, 39

[173] Siemens PLM. AMESim Website. `https://www.plm.automation.siemens.com/en/products/lms/imagine-lab/amesim/index.shtml`. Accessed: 2018-11-20. 141, 162

[174] B. Silver and B. Richard. *BPMN method and style*, volume 2. Cody-Cassidy Press Aptos, 2009. 45

[175] 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. 161

[176] 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. 44

[177] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *in Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001. 26, 27, 29, 30, 32, 37, 40

[178] Springer. Springer Link. `https://link.springer.com`. Acc: 2019-05-11. 23

[179] H. Stachowiak. General model theory. *Springer*, 1973. 11

[180] V. Stiehl. *Process-Driven Applications with BPMN*. Springer, 2014. 45

[181] V. Stolz. An integrated multi-view model evolution framework. *Innovations in Systems and Software Engineering*, 6(1-2):13–20, 2010. 31, 35, 39

[182] X. Sun. A Model-Driven Approach to Scenario-Based Requirements Engineering. `http://msdl.cs.mcgill.ca/people/simon/publications/thesis_singlesided.pdf`. 10

[183] SymPy Development Team. SymPy Website. `http://www.sympy.org/`. Acc: 2017-08-17. 92, 104, 162

[184] E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *Electronic Communications of the EASST*, 29, 2010. 168

[185] 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. 168

[186] TechSim Engineering s.r.o. LMS Virtual.Lab-Motion Brochure. `http://techsim.cz/wp-content/uploads/Brochure_LMS-Virtual.Lab-Motion.pdf`. Acc: 2018-08-17. 140

[187] The Jython Project. Jython Website. `http://www.jython.org/`. Accessed: 2018-11-20. 162

[188] The Open Group. The ArchiMate® Enterprise Architecture Modeling Language. `http://www.opengroup.org/subjectareas/enterprise/archimate-overview`. Accessed: 2018-11-20. 16

[189] 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. 44

[190] A. Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2), 2004. 168

[191] 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, 168

[192] W. van der Aalst. Business process management as the "killer app" for petri nets. *Software and System Modeling*, 14(2):685–691, 2015. 45

[193] W. van der Aalst and A. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005. 19, 45

[194] 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, 47, 106, 107, 110, 119, 156

[195] W. Van Der Aalst, K. M. Van Hee, and K. van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004. 17

[196] W. M. Van der Aalst. Process discovery: An introduction. In *Process Mining*, pages 125–156. Springer, 2011. 169

[197] R. Van Der Straeten. *Inconsistency Management in Model-Driven Engineering: An Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, Software Languages Lab, 2005. 30

[198] 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. 31, 36, 39

[199] 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. 31, 37, 39, 40

[200] 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. 169

[201] 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

[202] 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. 114, 120

[203] 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, 121

[204] Y. Van Tendeloo. Foundations of a multi-paradigm modelling tool. In *MoDELS ACM Student Research Competition*, pages 52–57, 2015. 121

[205] 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. 114, 121

[206] 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

[207] 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, 108, 113, 121, 162, 170

[208] Y. Van Tendeloo and H. Vangheluwe. Extending the DEVS formalism with initialization information. *ArXiv e-prints*, 2018. 107

[209] H. Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *CACSD. Conference Proceedings. IEEE International Sym-*

posium on Computer-Aided Control System Design (Cat. No.00TH8537), pages 129–134, Sep. 2000. 169

[210] H. Vangheluwe. An introduction to multiparadigm modelling and simulation. In *Proceedings of the AIS'2002 Conference 9–20 (2002). 35*. Press, 2000. 12, 13, 14, 26, 163, 165, 167

[211] K. Vanherpen. A contract-based approach for multi-viewpoint consistency in the concurrent design of cyber-physical systems. *University of Antwerp*, 2018. 14, 80, 148, 170

[212] 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, 144, 158

[213] 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. 161

[214] 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

[215] 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. 2

[216] 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. 81

[217] 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

[218] M. Weske. Business process management architectures. In *Business Process Management*, pages 333–371. Springer, 2012. 18, 167

[219] J. M. Wilson. Gantt charts: A centenary appreciation. *European Journal of Operational Research*, 149(2):430–437, 2003. 19

[220] 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

[221] 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

[222] E. Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011. 10

[223] 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, 121