SYSTEMATIC MODELLING OF EMBEDDED SYSTEMS:

MANAGING NON-FORMAL ASPECTS

Jelena Marinčić

SYSTEMATIC MODELLING OF EMBEDDED SYSTEMS:

MANAGING NON-FORMAL ASPECTS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. ir. A. Veldkamp,
on account of the decision of the Doctorate Board
to be publicly defended
on Thursday 21 September 2023 at 12.45 hours

by

Jelena Marinčić

born on the 24th of July 1972
in Belgrade, Serbia

This dissertation has been approved by:

Promotor
prof. dr. R.J. Wieringa

Co-promotor
dr. A.H. Mader

This research was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 612.063.407

Cover design: Guus Gijben, proefscrhrift-AIO
Printed by: proefschrift-AIO

ISBN (print): 978-90-365-5757-3
ISBN (digital): 978-90-365-5758-0
URL: https://doi.org/10.3990/1.9789036557580

Graduation Committee:

Chair / secretary:	prof.dr. J.N. Kok University of Twente, the Netherlands
Promotor:	prof.dr. R.J. Wieringa University of Twente, the Netherlands
Co-promotor:	dr. A.H. Mader University of Twente, the Netherlands
Committee Members:	prof.dr.ir. G.M. Bonnema University of Twente, the Netherlands
	prof.dr.ir. M. Boon University of Twente, the Netherlands
	prof. dr. C. Choppy Université Sorbonne Paris Nord, France
	prof. dr. J.G. Hall Open University, United Kingdom
	dr. ir. M. Verhoef ESA, the Netherlands

Abstract

Numerous (tool-supported) mathematical languages, formalisms and methods exist to create models and verify their properties. As their name suggests, formal methods for verifying and designing the systems focus on the formal, mathematical world. However, the steps to construct a model and to justify that it represents the system correctly with respect to its purpose are not entirely formal. Currently, a limited number of methods exist to guide through some of these non-formal steps. It is left to the modeller's experience and talent to learn how to construct and validate the model (justify that it correctly represents the system).

In this thesis, we argue that not all modelling steps, decisions and actions have to be unstructured and "out of the blue?. Steps, such as abstraction, can be explicitly named and placed in the modelling process. To date, these steps have been recognised, but not all of them have been gathered to form a modelling methodology that can help the modeller when designing the model.

The work presented in this thesis aims to improve embedded systems modelling by designing a method that complements the existing modelling languages and tools. The method focuses on non-formal, non-mathematical design aspects of software modelling techniques. We focus on modelling the software environment of high-tech, software- controlled systems consisting of mechanical and electronic components. Furthermore, we look at automata and state-based techniques.

We present a method that outlines modelling steps for constructing a model and justifying the model's adequacy. For constructing the model, our method provides the following components:

- A process to follow to design an adequate model and verify the requirements; at the same time this process can be used to explain how the model has been constructed, when performing model justification.

- Top-level reference conceptual models to frame the model construction and justification process. These conceptual models emphasise the gap between formal domain in which we can obtain formal proofs about model properties; and the informal world that we represent with the model.

- Structures including the following

  - The taxonomy of modelling decisions
  - Classifications of modelling assumptions

- A structure of the argument for justifying the model

The method can be used with other formal methodologies for modelling, as it aims is to complement them. Furthermore, the structures we provide aim to increase the knowledge of what the non-formal steps are when constructing and justifying the model.

Samenvatting

Er bestaan talloze (met gereedschap ondersteunde) wiskundige talen, formalismen en methoden om modellen te maken en hun eigenschappen te verifiren. Zoals hun naam al aangeeft, richten formele methoden voor het verifiren en ontwerpen van systemen zich op de formele, wiskundige wereld. De stappen om een model te construeren en om aan te tonen dat het het systeem correct weergeeft met betrekking tot het doel zijn echter niet volledig formeel. Op dit moment bestaat er een beperkt aantal methoden om sommige van deze niet-formele stappen te begeleiden. Het wordt overgelaten aan de ervaring en het talent van de modelmaker om te leren hoe het model te construeren en te valideren (te rechtvaardigen dat het het systeem correct weergeeft).

In dit proefschrift stellen we dat niet alle modelleerstappen, beslissingen en acties ongestructureerd en "uit het niets" hoeven te zijn. Stappen, zoals abstraheren, kunnen expliciet benoemd en geplaatst worden in het modelleerproces. Tot op heden zijn deze stappen erkend, maar ze zijn niet allemaal verzameld om een modelleermethodologie te vormen die de modelleur kan helpen bij het ontwerpen van het model.

Het werk dat in dit proefschrift wordt gepresenteerd heeft als doel het modelleren van embedded systemen te verbeteren door een methode te ontwerpen die de bestaande modelleringstalen en -tools aanvult. De methode richt zich op niet-formele, niet-mathematische ontwerpaspecten van softwaremodelleringstechnieken. We richten ons op het modelleren van de softwareomgeving van hightech, softwaregestuurde systemen die bestaan uit mechanische en elektronische componenten. Verder kijken we naar automaten en toestandsgebaseerde technieken.

We presenteren een methode die modelleerstappen beschrijft voor het construeren van een model en het rechtvaardigen van de geschiktheid van het model. Voor het construeren van het model biedt onze methode de volgende componenten:

- Een te volgen proces om een adequaat model te ontwerpen en de eisen te verifiren; tegelijkertijd kan dit proces worden gebruikt om uit te leggen hoe het model is geconstrueerd, bij het uitvoeren van modelrechtvaardiging.

- Top-level referentie conceptuele modellen om het modelconstructie- en rechtvaardigingsproces in te kaderen. Deze conceptuele modellen benadrukken

de kloof tussen het formele domein waarin we formele bewijzen over modeleigenschappen kunnen verkrijgen en de informele wereld die we met het model representeren.

- Structuren die het volgende omvatten
  - De taxonomie van modelleerbeslissingen
  - Classificaties van modelaannames
- Een structuur van het argument om het model te rechtvaardigen

Preface

Many years ago, while studying electrical engineering, I have never dreamed that models and modelling would have become a "red thread" in my career. Back then, I preferred to see the world described with formulae (which are also models of a kind). When professors presented their findings using block diagrams, I found these descriptions unattractive and wanted to see more details. However, as soon as I started working as an embedded software engineer and faced the complexity of real systems, stakeholder needs and different mental models of different engineers, I recognised the usefulness of abstraction and of different kinds of (informal) models. I started to use and never stopped, all sorts of different kinds of (software) models to communicate, design, verify and validate.

Through this thesis I was able to reflect on the process of modelling and perform research on topics I had already observed in practice, for example, how important it is not to forget what the purpose of a model is or to make sure that all the model reviewers have the same understanding of the problem. I am grateful to my promotor, Roel Wieringa and co-promotor, Angelika Mader, for taking me on this journey with them.

Furthermore, I am grateful to colleagues both in research and practice who have been sharing the enthusiasm for modelling techniques and tools and thinking about the process, reflecting what we are doing while modelling and why. Sometimes, in practice, when the goal is to deliver a product, there is no time for this kind of reflection. When we make the time for it, our models and designs become better.

Last but not least, I would not have been able to do this work without the support of my family and my circle of close friends that feel like family. Thank you so much!

Contents

Chapter 1

Introduction

1.1 Embedded software in the problem world

Software controlled systems are so deeply integrated into our lives that the days when machines, vehicles and devices had no or very little software seem like ancient times. Only a few decades ago, all these systems consisted mainly of mechanical, electrical and electro-magnetic parts. Nowadays, it is the software, called *embedded software*, that controls system components.

Embedded software enables complex system behaviour, high performance and connections to other software-driven systems. As today's machines and devices become more sophisticated, the complexity of their software rises enormously. If software complexity is not managed, the software becomes difficult to understand, verify, maintain and evolve. This, in turn, results in unfulfilled system requirements, project delays or failures and, in extreme cases, system behaviour that threatens their users' safety.

Software ensures that the system as a whole – be it a machine or a smart device – delivers the desired behaviour and the intended functionality. Therefore, one of the first steps in managing *software* complexity is understanding and managing the *system* requirements. For example, a luggage handling system is designed to route suitcases to correct conveyor belts and to move the conveyor belts until all suitcases are picked up. To achieve the required functionality, software ensures that a sequence of tasks is performed by different system components in a coordinated way: in a luggage handling system, it coordinates how belts move and opens mechanical switches to keep suitcases on their desired route.

The luggage handler is an example of a *reactive system*, with mechanical and electrical components reacting to events in its environment. From the perspective of software design, the suitcases, passengers, conveyor belt and mechanical switches all belong to the *problem domain*. Software transforms this problem domain into the desired solution domain, in which passengers get their suitcases efficiently, using conveyor belts. To be able to control and change the domain,

software constantly monitors, adjusts, fine-tunes or changes the problem domain in real-time. This can only be achieved by correctly representing the problem world internally in the software. For example, a software controller may be designed to keep track of mechanical-switch positions, the speed of the belts, and presence of the suitcases.

To represent the problem world correctly in the software, the designer needs to understand the problem domain. Typically, the problem domain cross-cuts different engineering disciplines. Engineers and domain experts use different mathematical frameworks and specify the system's behaviour using mathematical equations, informal notations and a domain-specific vocabulary. The software designer translates these specifications into software, and the art here is to make sure that this translation is accurate and at the same time not unnecessarily complex.

The software that manipulates complex domains and delivers complex behaviours reflects the domain and system behaviour, and thus will also reflect the system's complexity. This is an inherent, necessary complexity that cannot be avoided. However, to ensure good-quality software design, one should not add unnecessary complexity.

In today's world, where we heavily rely on increasingly complex systems that need to perform safely and securely, designing software that we can trust is crucial. Designing and, equally importantly, verifying trustworthy and dependable systems is much easier if engineers do not have to manage unnecessary complexity.

In this thesis the **scope** is **software-implemented embedded controllers** of **mechatronic systems**. Mechatronic systems are ones with mechanical and electrical parts and software as their main components. This thesis uses the terms "mechatronic systems", "cyber-physical systems" or "high-tech systems", interchangeably.

1.2 Model-based software design

Model-based software design focuses on software specifications at an abstraction level that is higher than code. Software models describe software in terms of the resulting system behaviour and properties. For example, they describe system actions, the order of those actions, synchronisation events and information that subsystems exchange. All the details of the code specific to a chosen programming language, operating system, software infrastructure and platform are separated, as they become an implementation of the model-based specification.

Using models in software design provides many benefits. First, models are easier to understand and manipulate than a more detailed implementation, in which the code details obfuscate the reasoning about the system behaviour. Both the designer and stakeholders can concentrate on their interests without being overwhelmed by unnecessary details. Typically, stakeholders are experts in their own domains and express requirements in terms of what the system

needs to do for end-users. The models serve to bridge the gap between requirements and software implementation (Brambilla et al. 2017).

Second, model-based techniques often come with formal and automated model analysis. This provides a way to verify and validate the model. The first process—verification—checks if the modeller described what they think they described. As described by Thacker et al. (2004), verification determines whether a model represents "the developer's conceptual description of the model and the solution to the model". These authors also Thacker et al. (2004) define validation as the "degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model." Schumann & Goseva-Popstojanova (2019) contend model based software engineering requires verification and validation (V&V) of the code as well as model itself. The following activities are performed: model inspection and review, static model analysis, model simulation, formal model analysis and model-based testing. The first activity is performed by the modeller together with domain experts, and the rest are supported with formalisms and tools.

Third, many modelling tools allow transformations to different between modelling languages, including the code (Brambilla et al. 2017). This reduces the chance of translation errors and improves the software's quality (Hutchinson et al. 2014) (Liebel et al. 2018).

Fourth, the goal of model-based software technologies is also to improve the efficiency of software design. Whereas some studies, such as the one of Hutchinson et al. (2014), report that the way that modelling increases the efficiency of code generation offsets required, additional investment, others are inconclusive (Liebel et al. 2018). What we can state is that, when technical and non-technical hurdles are overcome, modelling techniques improve software-development productivity.

Last, an important advantage of using models is one common to models in all engineering disciplines: we can learn something about the system by looking at the model, without having to prototype, change or manipulate the system first (Simon 1996). This allows for verification in early development stages, which saves costs of software debugging and integrating into the system after it has already been built.

1.2.1 A note on the industrial adoption

While a broad spectrum of model-based methods, tools and techniques have been around for decades, they have not become the standard way to develop software in the industry (Felderer et al. 2018)(Bucchiarone et al. 2020). Various authors agree that there are both technical and non-technical factors, and break it down in different ways. Whether we call those non-technical factors, cultural, social and economic (Selic 2012), social and community (Bucchiarone et al. 2020), organisational and managerial (Hutchinson et al. 2014)—the point is that technical advancement only is not enough. To be adopted in industry, both technical and non-technical obstacles have to be addressed.

1.2.2 The focus in this thesis

Software experts often use the term model-based software design or model-based software engineering as a common name for a wide range of modelling techniques, languages and tools. These techniques differ in their ability to generate code automatically, in their degree of formality, existence of precise semantics and in the languages they use. They are created to address different types of problems, thus the differences between these techniques. The variety of techniques, tools and the problems they are addressing makes it difficult to derive a single classification.

In this thesis, we will focus on modelling of system components' behaviour using **state-machines and automata** concepts. These models describe dynamic aspects, such as system actions controlled by software, the order of those actions and their timing.

1.3 The need for modelling methodology

In model-based software design, a model becomes the main design artefact, which is a specification of the implementation in the code. Designing software with models instead of coding represents a fundamentally different design approach. To learn to model, one needs to learn more than a syntax of a new language or how to use a tool. Model-based software engineering requires new concepts and new ways of looking at the design problem. In one word, designers need a new (modelling) methodology. And with a new methodology, comes a new way of thinking.

One could, of course, argue that learning a new programming language also comes with new concepts, or a different way of dealing with existing ones, and that learning it requires a change of perspective. Nevertheless, learning a new language does not take a designer out of the comfort of an existing methodology. For example, structured programming comes with its concepts of nested blocks, loops and points of choice (if-statements), and if developers switch from one language to another, they will only need to learn the syntax of a new language, and understand slight semantical differences.

It gets more difficult when a designer knows structured programming and is learning object-oriented (OO) design. In this case, the designer will need to drastically change their view on a design problem. There will be objects, kinds of objects, aggregates and other new concepts that do not exist in structured design. Again, once the designer is comfortably settled in the OO world, empowered with the knowledge of design patterns, switching from one OO language, such as Java, to another one, such as C++, will not be trivial, but it will not require unlearning old methodological concepts and learning new ones. A shift from programming to model-based software engineering requires a methodological shift.

Introduction to new modelling techniques will start with this premise: the books, courses, lectures and articles about modelling languages and tools will

often start with the statement that modelling requires special skills. They also recognise that 'beautiful' models are discoveries—inventions resulting from a creative activity. But, this creativity cannot be described with mathematical frameworks, in such a way that helps make better designs. So, while a novice modeller can partly learn the skill of modelling through formalisms and mathematics, the creative part is mastered only through experience. In the best case scenario, books that teach modelling partially address the informal parts of modelling.

Indeed, the skill of modelling is creative, deriving from experience and talent. There is no complete cookbook, checklist, or framework that a software engineer can read or learn to instantly become a good modeller. As any design activity, it takes time and experience. The novice in modelling has to try different modelling solutions, to make mistakes, to create models that are overly and unnecessarily complex, to develop models that are incorrect and to produce models that look 'ugly'.

However, we argue in this thesis that not all modelling steps, decisions and actions have to be unstructured and "out of the blue". Also, there are steps, such as abstraction, that can be explicitly named. And once they are named, we can look at their place in the modelling *process*. To date, these steps have been recognised, but not explicitly gathered in the form of a modelling methodology. Our approach is to name the steps, to find their place in the modelling process and to make them explicit. By making them explicit, we will explain to a (future) modeller what they are doing when modelling so that they can learn and mastering the skill.

1.4 The goal of the thesis

The goal of the work presented in this thesis is to improve **embedded systems modelling** by designing a *method* that supports using existing modelling languages and tools. The method focuses on non-formal, non-mathematical design aspects of software modelling techniques.

We focus on modelling the software environment of high-tech, software-controlled systems consisting of mechanical and electronic components. We elaborate the details of such systems in the next chapter. Furthermore, we focus on the environment behaviour described using concepts of automata and its variants. We look at both formal and non-formal modelling techniques that are state-based.

1.4.1 Parts of the modelling process on which we focus

They parts of the modelling process on which we focus is as follows.

(1) *Model construction:* identifying and modelling relevant problem domain concepts.

(2) *Model justification:* building an argument that the model adequately represents the system.

1.4.2 Model construction

When constructing a model, the modeller makes design decisions how to describe the problem world. This is a creative step and there is more than one correct outcome.

We analyse the modelling process and look for reusable, repeatable steps, in the generic sense—not specific to a particular domain or a formalism. The goal is to name modelling steps, to make them explicit, to be able to refer to them when designing a model.

1.4.3 Model justification

A model, as a deliverable, is a design product on its own. It needs to be assessed whether it correctly represents reality and is fit-for-purpose. This assessment is not possible using only mathematics. What can be done, however, is to compose an argument that increases our confidence in the model. The literature does not indicate how to do this systematically.

If the modeller possesses the knowledge of all relevant system details, then they can model their own understanding of the system. Additionally, they can verify the model, using tools and techniques at hand. However, in practice, the modeller is not necessarily a domain expert or the end user of the model. Therefore, the modeller mustassess the model together with the domain experts. The modeller explains the model to the domain experts and checks if their understanding of the problem domain is correct. If the problem domain changes, or for later use of the model, it is useful to capture this explanation in a structured justification argument.

Note that we use the term “justification”, and not “validation”. The reason is that the term “validation” in the world of software engineering typically means a formal process. To increase the confidence in the model’s adequacy and correctness, we cannot be entirely formal (mathematical). Inherent in this process is an informal aspect. Therefore, we use the term “justification”.

1.5 Design problems and knowledge questions

The top-level goal of the thesis is to design a method that improves the construction of models of cyber-physical systems. We focus on the model construction steps and on building a justification argument that the model is correct. This top-level goal has been broken down into design sub-goals and knowledge questions, shown in Table 1.1 (together with the chapters in which they are addressed.)

Design problems and knowledge questions
1.1 Design problem (addressed in **Chapters 5, 6 and 7**) Design a method that guides the **construction** of formal models of cyber-physical systems - that is independent of the formalism (modelling language) and - that supports communication among model users about the system and its model. **1.2 Knowledge question** (addressed in **Chapter 10**) To which type of cyber-physical systems can this method be applied?
2.1 Design problem (addressed in **Chapters 5 - 9**) Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism (modelling language) and - that supports communication among model users about the system and its model. **2.1.1 Design sub-problem** (addressed in **Chapters 9 and 10**) How to organise justifications to facilitate communication among different domain experts?

Table 1.1: Top level design problem is broken down into design problems and knowledge questions.

1.5.1 The research method at a glance

We explain the research method in detail in Chapter 3. In short, we first performed a literature review on the state-of-the-art on the non-formal steps of model construction and model justification. We then created the first version of the method and applied it to a case study. Based on insights from this initial case study, we adjusted and extended the method. We then applied the method to three other cases. In these cases we confirmed the model construction steps of the method and expanded the library of justifications.

1.6 The deliverables

The deliverables of the thesis are prescriptive and descriptive. The prescriptive ones are contained in the thesis design goals. They are:

(1) The method for a structured, systematic model design.

(2) Guidelines for building a semi-formal argument for model justification.

(3) A technique to collect and analyse assumptions made while modelling. The assumptions are conditions under which the model is correct for the purpose at hand.

Another deliverable of the thesis is an analysis of non-formal modelling aspects and related problems. The value of this analysis is that it makes implicit modelling steps explicit, which can help when teaching others how to model.

1.7 Thesis outline

In Chapter 2, we define terms and concepts used throughout the thesis. This chapter includes definitions of "mechatronic system", "control" and "plant", the terms that are used differently by different domain experts. We list different modelling methods and types of models on which we focus. Finally, we introduce the conceptual model that is the starting point and reference model for the thesis.

In Chapter 3, we explain what research methods we used. This research falls into the category of design science and we distinguish between design goals and knowledge questions.

In Chapter 4, we explore related work. We identify what non-formal aspects of modelling are addressed in existing software modelling frameworks and techniques. We also refer to a small number of articles that combine different techniques, and how they proposed to design models.

Chapter 5 outlines the initial version of our method. It is based on a conceptual analysis of modelling in general. It presents non-formal steps, decisions and aspects already treated by different communities, plus our own analysis and classes. It serves as a basic theory of non-formal modelling steps that we will use throughout the thesis.

Chapter 6 describes the first case study we performed to test the first version of the method: a Lego sorter. We chose a simple, but realistic, embedded system to explore the issues related to non-formal modelling decisions, plant modelling, and the relationship between formal an non-formal steps.

Chapter 7 describes our first case study performed in an industrial setting. The case study includes modelling of a paper inserter. The case discusses the reuse of modelling knowledge for mechatronic systems.

Chapter 8 covers our second industrial case study - modelling an industrial printer. In this case, the domain expert is not fluent in the modelling language used, and the justification of the model was the central point.

Chapter 9 focuses on specific model justification aspects, including modelling assumptions. It contains examples from the first three cases as well as a new case of a robot cart.

We then integrate our findings in Chapter 10.

Finally, Chapter 11 discusses our findings, their applicability, generalisability, and questions that remained open.

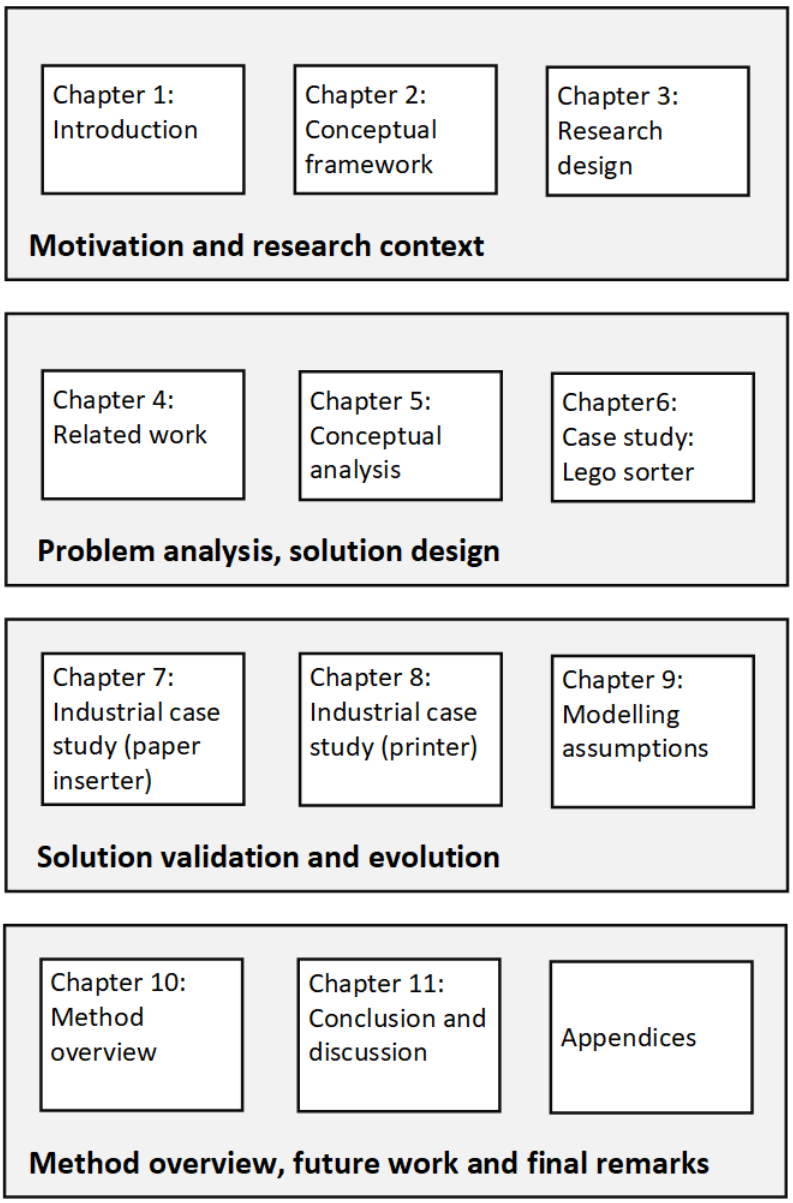


Figure 1.1: Overview of chapters mapped to problem-solution-validation structure.

1.8 Intended audience

Practitioners using formal methods.

Ever since Hall (1990) dismissed the 'seven myths' of reasons not to use formal methods in practice, formal methods researchers have explored why practitioners do not always fully embrace these methods. Bowen & Hinchey (1995) came up with more 'myths'; many others, including Bowen et al. (1996), and Hall himself (1998, 2005, 2007), continued advocating the application of formal methods in practice.

Our work adds another important aspect to the formal methods application area, namely modelling steps that have been implicit and modelling elements that are not formalised. Our method, designed to guide the modeller, addresses these aspects and complements existing formal techniques and methods. As such, it is intended for both researchers and practitioners who are interested in applying and adopting formal methods in industry. The latter community is often left with techniques that are conceptually proven, even supported by tools, but the techniques do not address all the context surrounding them when used in practice. The modelling elements that we addressed are part of this context.

Practitioners using model-based and model-driven methods.

The model-driven design community has developed various methods to perform different model transformations. We offer our method as a complement to this work, as we deal with those steps that cannot be transformed or automated.

Requirements engineering practitioners and researchers.

Describing the system with a verification model is inseparable from describing the requirement to be verified. Almost all the steps performed while modelling concern directly or indirectly requirements: while decomposing the system, we decompose the requirement as well. When we decide how to describe a system aspect, we ensure that it fits with the verification query that describes the requirement, and so on. Thus, our work may also interest requirements engineering researchers, in particular those interested in Jackson's (2000) problem orientation in software specification.

Researchers and philosophers of science explaining modelling of software systems.

To be able to understand the problem, we conceptually analysed the modelling process. We started from some known philosophical concepts that stand for models in general, but we applied them in a narrow area and identified what is missing in these concepts that is useful to understand modelling process. As such, they give some insights to the philosophers as well.

Chapter 2

Conceptual framework

In this chapter we introduce the conceptual framework of the thesis. We introduce the following concepts and related definitions.

1. Mechatronic systems along with their plant and software-implemented Controller
2. The plant and system's environment as the problem domain
3. The kinds of verification models in our scope
4. The system-model relation (conceptual model)
5. The problem definition, and how it is positioned in the conceptual model

2.1 Mechatronic systems

Our systems of interest are **mechatronic systems**. They consist of mechanical, electrical, electronics and software components. They require a multidisciplinary approach to design and verify them, hence the name. In this thesis, we use this term to distinguish from other systems, such as information systems that do not have electro-mechanical components and are mostly implemented by software.

A mechatronic system's components are **physical** components that move, interact and perform their own local function to deliver the global system function. Furthermore, many mechatronic systems deliver their main functionality by manipulating **products** or **workpieces**. These products move through the system, where they are being measured or in some way transformed. One such system we mentioned in Chapter 1—a luggage handler that moves suitcases along mechanical conveyor belts. Other examples of mechatronic systems that move workpieces include the following systems.

- A pick-and-place manipulator that moves, sorts and forwards bottles towards filling and labelling stations.

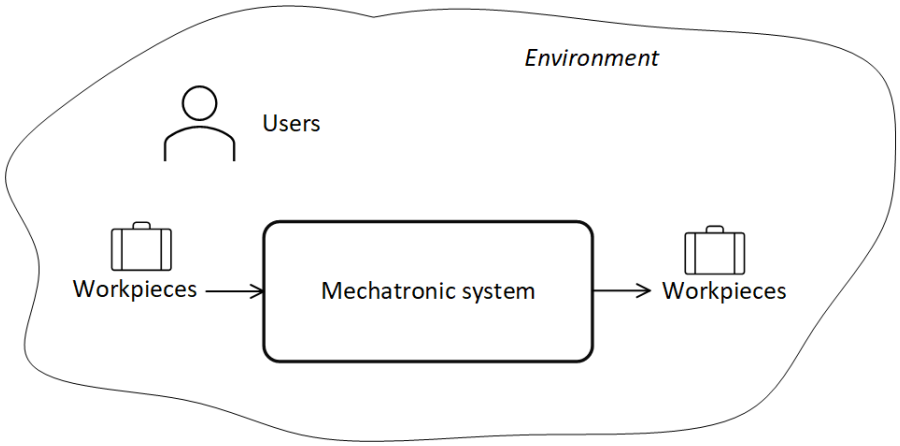


Figure 2.1: A mechatronic system and its environment.

- An MRI scanner that scans a patient and creates medical images (in this case a person is the "product" being measured by the system).
- A lithography machine projecting light through a reticle onto silicon wafers (in this case, we have two kinds of workpieces - a reticle and a wafer).

The products (workpieces) are not an integral part of the system, in the sense that they are not one of the building blocks of a machine or device. They are, however, a relevant part of the machine's **environment** and important to consider (and model) when designing the system. Figure 2.1 sketches a luggage handler in its environment. Apart from the suitcases (workpieces), there are **users** of the system. In other cases, other machines may interact with the system, so these other machines are also relevant elements of the system's environment. It may be relevant to represent other system stakeholders, for example a service engineer.

We could have considered the workpieces as an integral part of the system and incorporated users into the environment. What is important here is the notion of the relevant environment—those elements of the system's surroundings that need to be considered when designing the system. These relevant elements are determined by the system **requirements**.

Our focus in this thesis are the **Controllers** of these systems, typically implemented by the software. The Controller coordinates and integrates the behaviour of mechanical, electrical and other components, so that they deliver the desired functionality at the required quality level. We refer to the rest of the system, which consists of mechanical and other components, as a **Plant**. Figure 2.2 sketches the luggage handler in its **environment**, with the Plant and the software-implemented Controller. (Note that this informal diagram separates the software from the rest of the system for the purpose of defining the term Plant. This is not necessarily the first system decomposition that we

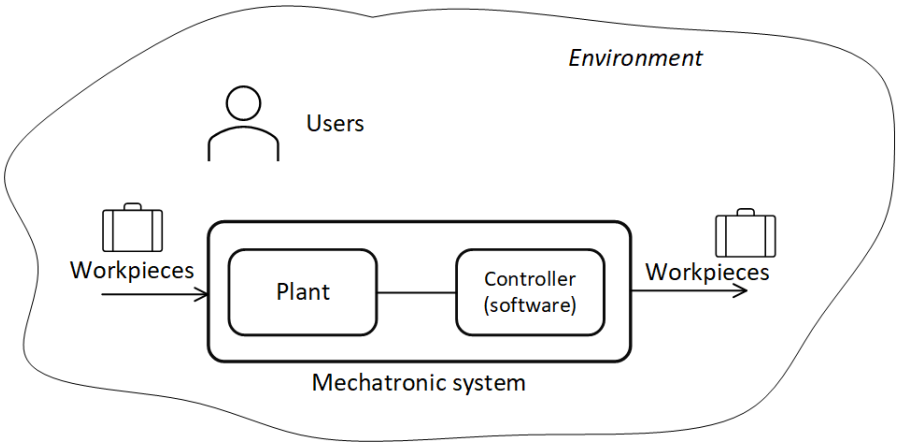


Figure 2.2: A mechatronic system decomposed to the Plant and the software implemented controller.

recommend to make when modelling and designing the system. It may be more convenient to divide the system into sub-systems and then separate the sub-systems' controllers from the physical and electrical components they control.)

The Controller in our focus is a "high-level" (process) controller that coordinates the system's physical parts. For example, it controls steps in a process inside the system, by starting, moving and stopping mechanical components. The Controller tracks many physical parameters of the plant and steers it accordingly. Also, it stores pieces of information about the Plant for when they might be needed later. Essentially, the controller internally represents the knowledge about mechanical, electrical parts for the purpose of controlling them. For example, a product and its position relative to other physical components is represented by values in a coordinate system, which is also symbolically represented in the software. This internal representation of physical behaviours and properties requires the designer to understand, integrate and model the system aspects designed by other engineer disciplines, such as mechanical, electrical and chemical.

To summarise, we focus on the following mechatronic systems' aspects.

- The system as a collection of physical parts that move and perform individually, to deliver together the overall functionality
- Workpieces that enter and leave the system after being transformed and/or measured
- The system boundary, the system environment and the environment boundary
- The system as a Plant controlled by the software-implemented Controller
- The Controller that controls the system processes (a high-level controller)

Naming conventions for this thesis	
Mechatronic system (alternative names)	**Controller** (alternative names)
System	Software (implemented) controller
Cyber-physical system	Embedded software
High-tech system	Software

Table 2.1: Terms used interchangeably.

A note on naming Our systems of interest are *mechatronic systems*. In the literature, these systems are sometimes called *cyber-physical systems* while some industry practitioners refer to *high-tech systems* (for complex machines sold to other businesses, as opposed to consumer products). We use these terms interchangeably throughout the thesis. The Controller implemented in the software is often referred to as **embedded software**. We will also use the terms **software (system)**, **software controller** and **controller** interchangeably. We summarise this in Table 2.1.

When dividing the system into the software implemented controller and the physical plant, we abstract away (1) embedded system hardware on which the software runs; (2) the software implementation related aspects; (3) the control engineering aspects of the Controller; and (4) the social-technical aspects of systems. We look at the control specification that directly "talks" to the Plant.

For the remainder of this section, we explain in more detail these abstractions, and how they relate to our focus.

2.1.1 Abstracting away embedded system hardware

Embedded software runs on an embedded system. Embedded systems are special-purpose computers inside mechatronic systems as well as other devices and pieces of equipment. An embedded system consists of a hardware part—which is an integrated board with processors, memory and other hardware elements—and the embedded software. In some cases embedded software runs on a general-purpose computer and the computer is placed somewhere physically close to the machine, or far away, connected via a network. In this thesis we will abstract away the hardware part of an embedded system and consider only the software.

Embedded software is a typical reactive system that observes, acts upon, or reacts to its environment. It interacts with its environment through sensors and actuators. Often, the system is designed with a user interface that lets the operator choose among different plant behaviours. Furthermore, the system can interact with (interface) other systems.

A sensor is a device that converts different physical phenomena, such as temperature, humidity, speed and pressure into electric signal. Usually the

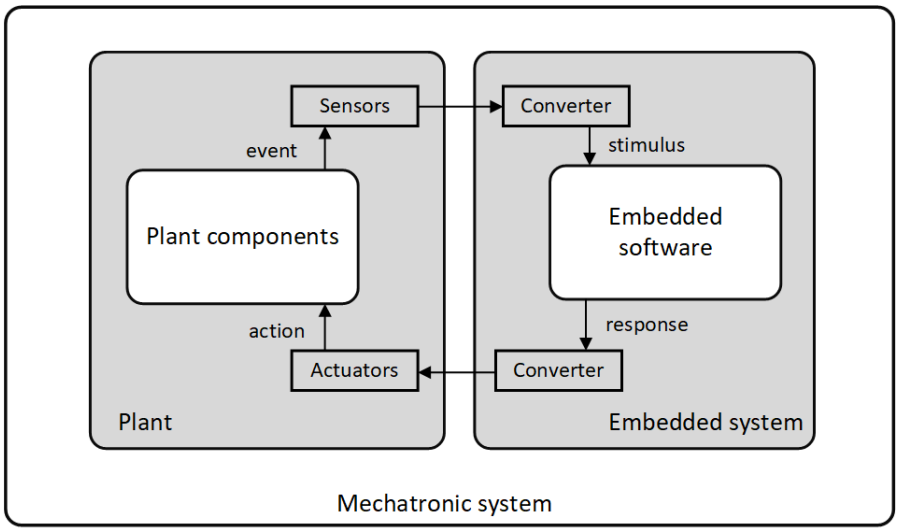


Figure 2.3: The details of how software Controller controls the Plant.

electrical signal of a sensor is analogue and needs to be converted into a digital signal so that software can read it. For certain phenomena there is no sensor, but the control software calculates an estimation based on the data from another sensor or based on some previously defined calculation.

An actuator converts anelectrical signal into motion or some other Plant phenomenon such as heat or light. It can mechanically convert one type of motion to another, or it can be used to prevent motion. A typical actuator is a motor that transforms electrical current into mechanical energy, namely motion with a desired force applied to the object connected to the motor. The speed and force achieved are a function of the electrical current applied.

Figure 2.3 zooms into the mechatronic system shown in Figure 2.2 and elaborates the relationship between the Plant and the software-implemented Controller. The Controller observes and actuates the Plant through sensors and actuators. Sensors observe (measure) the state of the Plant and their signals are converted into stimuli for the software (controller). The Controller actuates the Plant by sending a response that is then converted into a signal for the actuators, which in turn acts on the Plant. (Many software frameworks only talk about the "events". In this picture, we distinguish between events, actions, stimuli and responses as it is done in Wieringa (2003) so that we can show a number of abstraction steps made when talking about events in the software that represent event in the Plant.) Note that in this figure we have one block for the embedded software and one for the Plant, whereas in practice multiple software sub-systems or components control multiple Plant components.

We consider sensors and actuators as part of the Plant, in most of the cases we consider them integral parts of other plant components. For example, a

motor is part of the conveyor belt. In some cases, it is relevant to take into account (model) sensors and actuators explicitly. One such example is a complex system in which the sensor itself is a complex sub-system.

In Figure 2.3, the greyed out blocks are those that we abstract away to end up with the blocks Plant and Controller shown in Figure 2.2. The communication path through sensors, actuators and (signal) converters in Figure 2.3 are abstracted away and become one communication line between the Controller and the Plant in Figure 2.2.

2.1.2 Abstracting away software implementation related aspects

In the previous section we described how embedded system hardware is abstracted away and represented by the embedded software block. We also described how the Plant is represented by its components without modelling the sensors and actuators. And we described how this also means that communication lines between abstracted elements become the communication between the embedded software and the Plant. In this subsection, we describe how the software's implementation details are abstracted away and how embedded software is represented as the high-level process Controller.

Figure 2.4 zooms into the embedded software block shown in Figure 2.3. The embedded software that implements the control performs the following tasks.

- It converts the electrical signal coming from the sensors and converters into information about the Plant. It also translates the values from the feedback controller into an electrical signal for the actuators. In the diagram we named this component the *Drivers*.
- It implements a specification of a closed-loop or open-loop controller of the Plant. This specification is described using control theory. In the diagram, this block is called a *Control algorithm application*. This controller regulates the dynamical aspects of the Plant.
- It coordinates system components and it controls the overall system workflow. It is the high-level control that we already mentioned as being our focus. In the diagram the block is called the *Controller*.
- It communicates with the user or surrounding systems via a special interface. In the diagram the components performing this task are represented with one block named the *User interface communication*.

It is standard practice to separate the above elements in the software design and architecture. One possible software architecture pattern is shown in Fig. 2.4. It recognises these separate components as software layers, each in charge of their own context. Communication between the layers is reciprocal, but the arrows go only in one direction to represent the following: an upper layer does not "know" all the implementation details of the lower layer.

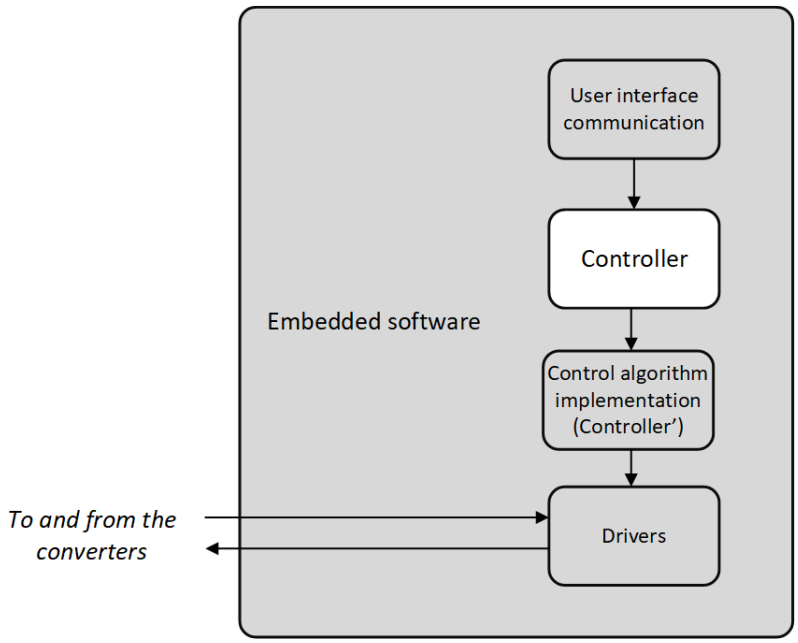


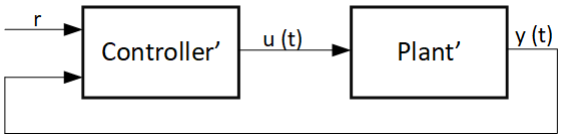
Figure 2.4: Layers and modules of the software that 'see' the Plant on different abstraction levels.

Other software architecture patterns besides the layered one also exist. But, while designing the software, the above-mentioned elements are described, specified and modelled separately before being integrated. In this thesis, we abstract away these implementation details and focus on the Controller. In our representation, the Controller communicates directly to the Plant. We look at how control engineers and software engineers communicate so that they can perform this abstraction step correctly. For this reason, in the following subsection (2.1.3) we explain the basics of the control engineers' view at the Plant and what they call a controller; and in the subsequent subsection (2.1.4) we describe how the software engineer typically views the Controller.

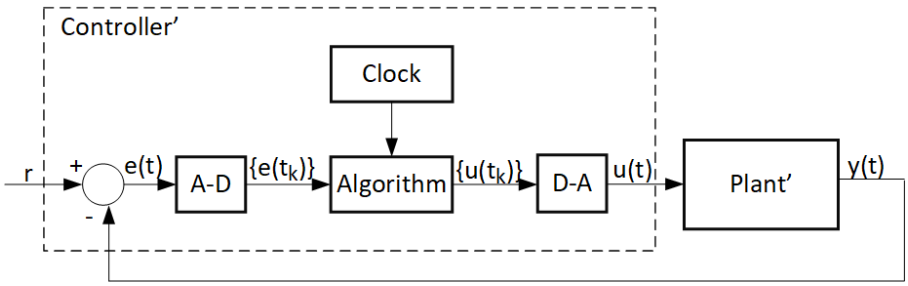
2.1.3 The control engineers' control

In Figure 2.4, the *Control algorithm implementation* block implements the mathematical description of the system control, derived from the conceptual and mathematical framework used by the control engineers. The controller designed within this framework controls the dynamical aspects of the Plant, which is a behaviour that changes over (as a function of) time (van Amerongen 2010).

For control engineers the problem starts with the desired plant behaviour from the users' point-of-view. One example, described by van Breemen (2001) starts with the user requirement to keep a house's temperature comfortably sta-



(a) The plant and the control (control engineering view)



(b) Inside a (digital) Controller

Figure 2.5: Control engineering view on system: the Plant describes the dynamic aspects

ble, using a heating system with a controller (thermostat). The control engineer will translate the top-level requirements into one or more control engineering problems. The controller is designed within the system theory mathematical framework, but is unknown to mathematical framework of the software.

The mathematical equations and models that describe the dynamic elements of the plant and its controller are represented graphically by a block diagram in Figure 2.5(a). This diagram shows a feedback-controlled system, in which the present state depends on past behaviours (van Amerongen 2010).

The plant's controlled variable is y: in our example, the room's temperature. Its reference value r is provided as an input to the Controller' in our example, the desired room temperature. The heat loss is not easily measured or predicted; therefore, the output temperature is measured and brought back to the controller. The feedback mechanism enables the controller to keep correcting the temperature, based on the difference between the desired and output temperature. The control signal u ensures that y stays within the margins of the reference value r. The control signal u is the output of the controller and input to the Plant: for example, it can be the value of the current that actuates the heating sub-system.

The block diagram, such as in Figure 2.5(a) graphically represents mathematical equations that model the Plant and the Controller, independently of the domain (for example, a thermal, hydraulic, or electric phenomena). As explained by van Amerongen (2010) the block diagram unambiguously represents

the controller model and the plant model at an abstraction level higher than the underlaying equations and it outlines their causal relationship.

The controlled system parameters and processes change over time due to disturbances in the plant and its environment. Therefore, the output is measured in real-time and the control signal is adjusted accordingly, also in real-time, based on the controlled variables' measured values. In Figure 2.5(b) we show an example of a digital controller controlling continuous output. The error signal, based on the reference value and continuous output signal, is converted into a digital value in analogue using a digital (A-D) converter, with period T, controlled by the Clock. The signals are measured in discrete moments t_k, t_{k+1}, and so on. The digital controller, implementing the control algorithm produces a digital control signal, also in discrete moments in time. The digital to analogue (D-A) converter transforms it into an analogue signal.

The digital controller implements a control law, a mathematically described algorithm that defines the necessary dynamics of the input signal in order to achieve the desired dynamics of the output signal. The desired dynamics will keep the temperature, pressure etc. in the desired ranges and keep the frequency of their changes in a desired range.

The example in Figure 2.5 describes a solution to an elementary control problem. In practice, control problems are compound and decomposed into multiple partial control problems (van Breemen 2001). Typically there is no one measured variable and one control signal, but multiple ones. The input and output signals are typically voltages corresponding to measurements of temperature, humidity, pressure, speed, distance etc.

The control implemented by software periodically executes the following actions on signals described in Figure 2.5(b). It reads the measurement of the the output signal, compares it with the reference value, calculates the correction that will be applied to the input, and applies the signal value to the input of the system (Plant). The calculation of the correction is determined by control law. During the duration of the time between two clock ticks, the input signal stays constant. In the case of an open loop system, the input signal is applied under the assumption about the output signal behaviour.

In this subsection we described conceptual framework used to address control engineering related problems and requirements. This is the layer of the software we named "Control algorithm implementation (Controller')" in Fig. 2.3. For example the Controller in Fig. 2.3 will move a system component from point A to point B, and will start this action. How this is implemented, with the desired quality performance is the task of the layer below, the one we describe in this subsection.

2.1.4 The software engineers' control

Zooming out again, the Controller in Fig. 2.3 does not "know", does not describe dynamic aspects of the system. It uses a different mathematical framework to describe the system behaviour in terms of states and events. It responds to an event representing for example luggage being present on the conveyor belt, to

act upon it, and initiate luggage transport, which can be described as a new state.

To design this layer of control the software designer needs to analyse the requirements for the whole system and find how they can be achieved by the system components and processes. This is translated to the requirements for the system components. The decomposition is performed until the requirements for the software are defined.

The Controller also communicates with the user (interface). The phenomena in the Plant that the user wants to observe, either directly or remotely, is transformed into a form understandable for the user. Similarly, the commands coming from the user or the operator are transformed into the signals for the Drivers or Control algorithm layer.

2.2 Plant and environment as the problem domain

In previous section we sketched the Environment in Figures 2.1 and 2.2, on an example of a luggage handler. We also mentioned that the Controller internally tracks and describes the Plant and the Environment and, based on this information, "decides" how to actuate the Plant. One approach that structurally describes the role of these internal descriptions in the Controller, is the problem frames technique (Jackson 2000). It explicitly acknowledges the fact that the (descriptions of) Plant and the Environment belong into the Problem world and the software belongs to the Solution world.

To analyse the problem, before and while designing its solution, Jackson's (2000) problem frames technique suggests to start the analysis by separating the problem domain and its requirements from the (software) solution. Fig. 2.6 is a generic problem diagram used in problem frames technique to frame the software design process. This technique uses the name Machine for the software being designed and represented by the rectangle with double stripes on its left side. The plant and the relevant environment are represented as the problem domain (block "Domain" in the diagram). The requirements for the system describe the phenomena of the Domain. The requirements are represented with a dashed ellipse that refers to the Domain with a dashed arrow. The Domain and the Machine share phenomena (named a in the diagram) on their interface. The specification of the software (software requirements) describes the software behaviour in terms of this phenomena.

This simple diagram emphasises the fact that the system Requirements refer to the Domain, not the software. Further steps in this technique suggest to decompose the Requirements, the Domain and the Software requirements. Therefore, the one block representing the Domain in this picture can be further decomposed to sub-domains that can also share phenomena with the Machine or among themselves. We will show how we use this technique in our method in later chapters. In Figure 2.7, we map the concepts we defined in previous

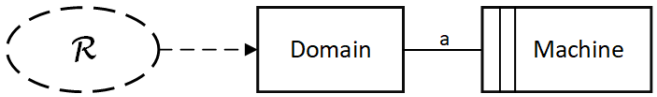


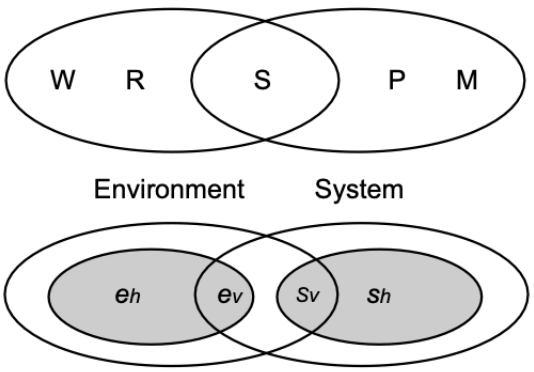
Figure 2.6: Jackson's problem diagram: software controller (Machine) controls the Domain so that it delivers its requirements.

Figure 2.7: Mapping our concepts of Controller, Plant and (relevant) Environment into Jackson's problem diagram

section into the problem diagram of problem frames technique. In the luggage handler example the problem domain for the requirement to transport and sort the luggage is the Plant itself as well as the luggage.

For semantics of the elements of Problem Frames, we refer to (Hall et al. 2005). For meta-models of Problem Frames that relate the framework to model-driven engineering (MDE) approaches we invite the reader to read the article of (Xiao et al. 2021) in which they give the overview of the existing meta-models and build-up with their own additions. For the formal descriptions of systematical steps of arriving from the domain requirements to software specification, we refer the reader to the article of (Li et al. 2014).

We introduced the mapping of our basic concepts into the problem frames concepts informally, but to provide clarity and preciseness we will describe them using Gunter et al.'s (2000) reference model for requirements and specifications. The goal of the reference model is to serve as a basis (reference) for formal description of user requirements and their reduction to the specification of the system behaviour. It is the underpinning of the problem frames approach, and it focuses on the phenomena shared between the system and its environment. As diagram on Fig. 2.8 shows, the system and environment are not isolated worlds but they share some phenomena.



W – environment description
R – user requirements
S – system specification
P – program
M – programming platform

e – phenomena visible to and controlled by the environment
e = eh ∪ ev, where
eh is not visible to the system and
ev is visible to the system

s – phenomena visible to and controlled by system
s = sh ∪ sv, where
sh is not visible to the environment and
sv is visible to the environment

Figure 2.8: The WRSPM reference model redrawn from (Gunter et al. 2000).

The reference model (also called WRSPM model) identifies five artefacts that refer to the system and/or its environment description. They are as follows.

- Domain knowledge about the system's environment (W)
- User requirements for the system's environment (R)
- Specification for a system that satisfies user requirements (S)
- A program that implements the specification (P)
- A programming platform (M)

Fig. 2.8 shows that W and R refer to the system environment, P and M to the system, whereas S describes the phenomena on the intersection of the two. These five artefacts are viewed as descriptions formulated in different languages. Whatever language is used, it refers to the classes of phenomena (states, events, individuals). The reference model is adjoined with the additional artefact – the designated terminology that provides names to describe the environment and the system (D).

Designations identify classes of phenomena (including events and states) some of which belong to and are controlled by environment (e), and some of which belong to and are controlled by the system (s). The environment phenomena at the interface between the environment and the system is visible to the system as well (it is denoted with e_v). The rest of the environment phenomena is not visible to the system (it is denoted with e_h).

Similarly, phenomena s_v refers to the interface of the system and the environment, it is controlled by the system and visible to the environment. The rest of the phenomena controlled by the system is not visible to the environment (it is denoted with s_h). Obviously: $e = e_h \cup e_v$ and $s = s_h \cup s_v$

W and R use the terms denoting phenomena in s_v, e_v and e_h as part of their descriptions. S uses only the terms denoting phenomena s_v and e_v to describe the specification. M and P refer to terms denoting s_v and s_h.

The system together with its environment should satisfy the requirement, or written in logic:

$$\forall es.W \wedge M \wedge P \Rightarrow R \tag{2.1}$$

If the process of the system design or verification is broken down to developing requirements and to their implementation. Then in the first phase we want the following to hold:

$$\forall es.W \wedge S \Rightarrow R \tag{2.2}$$

The next phase requires that there exist a P and M that implement S, but this is out of our scope.

2.2.1 Mechatronic system verification in WRSPM Terms

The WRSPM model is general and independent of the choice of language. Also, the WRSPM model leaves its users to decide where to draw the line between the system and its environment.

The WRSPM model elements applied in our case are as follows.

- The system in the WRSPM model is embedded control software of a mechatronic system.
- The system's environment consists of the the plant and relevant environment such as the products that the plant manipulates, and the system operator. We have described them in Fig. 2.1 and Fig. 2.2.
- The plant description describes the plant behaviour, which is the behaviour of its mechanical components ($W \equiv knowledgeaboutthePlant$).
- The user's functional requirement is already transformed into the requirement for the plant behaviour. Eliciting user requirements and refining them into the plant behaviour observable to the system user involves a considerable skill and is a research area on its own in requirements engineering. Here we assume that someone already prepared the requirement for the plant for us ($R \equiv Requirement$).
- The specification of the behaviour on the intersection of the system and its environment refers to the phenomena on the sensors and the actuators ($S \equiv Control$).
  - The actuators' states, events and values are the phenomena denoted as s_v - they are controlled by the embedded control software and visible to the environment.

- The sensors' states, events, and values are the phenomena denoted as e_v - they are controlled by the environment and observable by the environment.

On a higher decomposition level we can have

- the mechanical components or processes controlled directly by actuators can be treated as phenomena controlled by the system s_v (for example a stand in a microwave that is rotated by a motor; if we abstract the motor, the stand is then directly controlled by the system and its states are controlled by the system and visible to the environment) and
- the phenomena in the environment directly observed by sensors can be treated as the phenomena that is visible to the control, e_v (for example, movement of a pick-and-place robot arm 20 cm to the left, can be seen by the system (sensor is abstracted away))

System verification assesses whether

$$\forall es.Plant \wedge Control \Rightarrow Requirement \tag{2.3}$$

As already explained, we abstract away the Control laws of control engineers as well as the embedded controller hardware.

2.3 Model-based software engineering

Model-based software engineering proposes to create models to reason about the software and the system it controls, including validation and verification. Some modelling frameworks allow generating the code from the models. Model-based software design is a way to deal with ever-increasing software complexity. As described in Section 1.2, modelling can bring many benefits, but at the same time, for it to enter the main stream of the software engineering discipline fully adopted by industry, some technical and non-technical hurdles need to be overcome.

Despite the remaining hurdles for industrial adoption, model-based software engineering is becoming recognised as a branch of the software engineering discipline. Many "sub-branches" are named differently by different experts; also, there is a need to constitute fundamental concepts to define the discipline (Ciccozzi et al. 2018). In this thesis, we used the term in its broad meaning. When we talk about model-based software engineering, we have in our scope the models used for software Controller design and verification. They verify that the Controller and the Plant will satisfy the requirements. The requirements we focus on describe the Plant behaviour. In this section we briefly summarise modelling techniques used to model system behaviour that are in our scope. These are all the techniques that use state-based approaches to model the system. They are as follows.

- Model-checking
- Plant simulation models
- Software models used in model-driven software engineering described using state machines

2.3.1 Model-checking

Model-checking is used to prove software properties and functional requirements; see e.g. (Baier & Katoen 2008). Model-checking techniques are formal, which means that model properties are proven mathematically. Both the model and its properties are formulated mathematically. An important class of these techniques describes the system with automata and the model properties via a temporal logic. Automata describes the system with states and transitions; it is possible to check safety properties (the system never does something bad as some undesired state is never reached), liveness properties (from any point of time, the system eventually does something good, as some desired state is eventually reached) or fairness (will the system give a fair turn to its components). The verification algorithm implemented by the tool searches the model's state-space and produces the result with mathematical correctness.

The difference between (conventional) testing and model-checking is that testing involves developing a set of test cases and providing code coverage (a limited number of scenarios) whereas model checking determines if the model of the system satisfies specified properties by performing an exhaustive state-space search.

The tools implementing model-checking algorithms are limited by the processing power of the computers. When describing a system with multiple automata that run in parallel, there is a danger of state explosion—the computer running the algorithm that searches the state space may be too slow to explore millions of different states and paths. However, model-checking algorithms have been constantly improving (Clarke et al. 2012) to be more efficient while computer processing power has also been constantly increasing.

2.3.2 Hardware simulation models

Models used for software design do not necessarily only describe the software. Being a reactive system, embedded software responds to stimuli from its environment consisting of mechanical, electrical and other parts. Typically, when designing software for a new machine, these parts are unavailable, as they are being designed at the same time as the software. Software engineers have the specification of the machine's behaviour, but they still have to make numerous assumptions about the machine's parts. The same holds for engineers on 'the other side' – they have the software requirements, but they still have to make many assumptions about it.

If software and hardware are developed in isolation, with little communication between engineers, the integration phase becomes long and painful, when

it turns out that the software and the machine parts do not exhibit the expected behaviour. Too many assumptions will remain untested, undocumented and forgotten— and it takes a long time to find the root cause of any problems. In this late stage of the system's development, some of the problems can be solved only by changing hardware components, which is far more expensive than adjusting the software.

To support concurrent development of software and the machine's mechanical and other parts, the latter are modelled so that the software can be verified. This approach is also known as hardware-in-the-loop approach, see e.g. (Sarhadi & Yousefpour 2015). The models simulate machine behaviour, including the sensors and actuators, which work as the interface between embedded software and the machine. The software is plugged into the model as if it were a real machine. This way, software designers can discover whether some of their assumptions about the machine's components behaviour were wrong so that they can either fix it in the software or inform mechanical and other engineers about the changes necessary in their domain of work. The simulation model does not replace the real machine, but reduces the number of incorrect assumptions. At the same time, it stimulates communication between different domain experts in the early phases of the design, reducing the integration time and preventing problems that require expensive solutions.

Hardware simulation models often use state-machine based models, as this paradigm is known to all the engineers, not just software engineers. Sharing this paradigm enables communication between these engineers.

When it comes to formal semantics, most of the current techniques to simulate machine parts are not formal, meaning that there is no formal proof about any of the model's properties. Instead, simulation models require testing. There are approaches that recognise this limitation and work towards the frameworks that enable both simulation and verification, like for example (Bernardeschi et al. 2020).

2.3.3 Model-driven software engineering

Brambilla et al. (2017) define MDD (Model-driven development) as "a development paradigm that uses models as the primary artifact of the development process". They define MDE as the "superset of MDD" because it "encompasses other model-based task of a complete software engineering process". They define MBE (Model-based engineering) as "a process in which software models play an important role although they are not necessarily the key artifacts of the development (i.e., they do NOT "drive" the process as in MDE). MDE provides the techniques to automatically generate code from the model (Whittle et al. 2014). It is, therefore, not enough to make a high-level model of the system, but transformation models that will refine it to the code are also necessary. These transformation models are divided into those that are platform (operation system and programming language) independent and platform dependent.

Often, these approaches are used in combination with Domain-Specific Languages (DSLs). They are dedicated to a domain of interest (Clark et al. 2015)

and are semantically closer to the domain concepts. As such, they make the modelling task easier for the modeller (Brambilla et al. 2017) while also making the model easier to maintain and reuse. The investment into their development is justified when they are (re-)used for multiple products in a product line and platform development.

The state machines or automata are general-purpose modelling languages—an automaton that can be used to describe behaviour. In their taxonomy of MBSE approaches De Saqui-Sannes et al. (2022) place the "models relying on state/transitions paradigm" into category of formal languages, but note that that UML "grounds in Statecharts".

2.4 Model vs reality - the reference model

A model represents a system. According to Simon (1996), every model can be seen as a simulation of a system, meaning that we can learn something from the system by analysing its model. We will keep the term 'simulation' for those models that have dynamic, real-time behaviour. In models other than simulation ones that we use, such as automata-based models of system behaviour, we can also analyse the model and calculate or estimate or observe what the system behaviour is.In his Problem Frames book, Jackson (2000) calls models of the software environment 'descriptions', referring to the fact that models describe a system.

Philosophy of science uses terms 'representativeness' (Giere 2004) and 'target' to emphasise that any model is designed with a clear view of *what* it represents, a system, phenomena, part of the system etc. – a modelled target. The target may already exist or is yet to be designed, or does not exist and will not be designed at all; but the modeller defines first what is modelled, and then constructs the model. It can happen that one model is suitable to represent other targets, even if that was not intended. But the model's intended target is always defined in advance, before designing the model.

> Definition: A model is *adequate* if, by analysing or observing the model, we can draw correct conclusions about the system.

In software engineering, the target is a computer-controlled system, or one of its components. It can be software only or software in combination with its environment, a plant, operator, users; in information systems the environment will consists of people having different roles with predefined relationships between these roles, and of data representing users.

Defining the target system is only the very first step, the basic decision inherent to modelling. Other decisions that are crucial for the model's adequacy are the decision of what system parts and aspects exactly have to be modelled, and how precisely they have to be described.

The goal of modelling is to enhance, facilitate, enable system analysis by making it easier, less complex than observing the system directly, or having

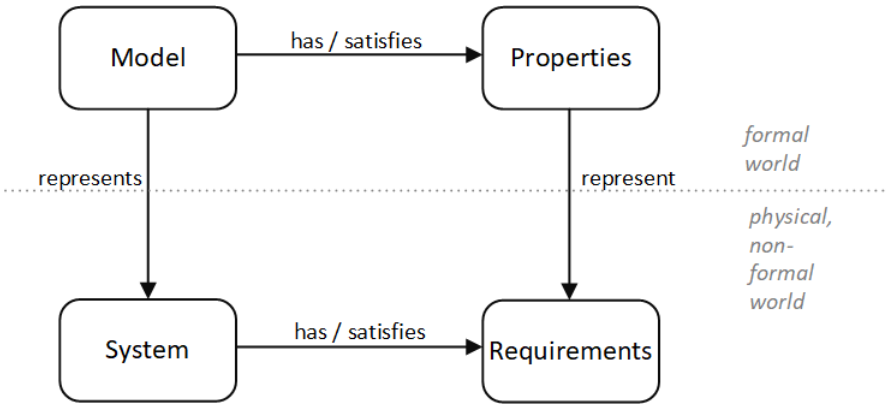


Figure 2.9: Relationships between the model and reality (the system that is modelled.)

to build it if it does not exist. The focus is narrowed down from all possible properties to a set of those that can be independently analysed. For this, it is necessary to define what these properties are, and what parts, components, aspects of the system are relevant for the analysis.

After they determine what is *relevant* to be described with the model, the modellers decide how precisely and on what level of abstraction the system components will be represented in the model. The goal is to have them described precisely enough to get correct results from the model analysis. Determining what has to be modelled and with what level of details is an iterative process.

In formal verification, the decision of what is relevant to model is guided by the system requirement in which we are interested. Figure 2.9 shows relationships between entities relevant in modelling for formal verification[1].

On the bottom half of the diagram are the system, and the requirement; they are the modelled targets. The model is designed to represent the system adequately with respect to the requirement that is analysed. The requirement is something that refers to the system, so it needs to be codified into the model property.

The model and the property belong to the mathematical world. Once they are designed, they are analysed, transformed, manipulated with mathematical tools. Whether the model possesses the property or not is determined with mathematical correctness. If the model possesses the property, than we conclude that the system satisfies the requirement. If not, we conclude that the system does not satisfy the requirement.

[1]Even though a variant of this diagram had been used in research discussions preceding the work on this project, it has not been published before that.

2.4.1 Formal versus non-formal modelling steps

Formal modelling steps are those steps that can be described mathematically in such a way that they can be automated and used for broad range of models. Formal verification methods and tools and model-driven formalisms and tools provide means for automated model analysis. After the modeller designs the model and formulates the property, an algorithm determines whether the model satisfies the property. Also, different transformations of models are automated. Having defined a modelling language, it is possible to formally describe these transformations, as well as state-space searches.

On the other hand, while designing the model, the modeller is performing many steps that cannot be formalised and automated. They include the system analysis, modelling problem analysis, deciding what has to be modelled and how. Researchers and practitioners do not focus on non-formal steps as a whole, because there are many levels, characteristics and processes to look at separately. Instead, modelling is analysed as a *process* consisting of general steps that are performed while designing a model.

2.4.2 Summary of modelling challenges

The challenges of embedded software modelling emerge from the following.

1. When designing or verifying software, it is necessary to describe the Plant and the system Environment without knowing in advance what parts, aspects and properties have to be described. There is no recipe how to do it. Creating a model is a design process. And designing any software artefact is not a straightforward process, but at the same time, is an act of discovery (Hall & Rapanotti 2008b).

2. The nature of system design is multidisciplinary. The Plant is designed by different domain experts in different disciplines. The modeller must distill and extract the necessary knowledge about certain Plant aspects. Moreover, the modeller has to translate this knowledge from the "language" used by the domain experts to the language of the model. Switching between the formalisms and frameworks of different disciplines is also not simple.

3. There is no mathematical, formal proof that the model represents the software and the physical world adequately.

The first two challenges hold for software design in general, not just for model-based design. The third challenge holds for all modelling pursuits, not just embedded software modelling.

The physical aspect

Even though software is responsible for the plant behaviour, it can only directly access motors and sensors. To be able to 'calculate', or 'reason' about the plant

parts connected directly or indirectly to the motors and sensors, there has to be in internal representation of the Plant within the software (Jackson 2000). It relates signals observed by the sensors with other monitored values in the plant, and values sent to the actuators with the behaviour of the plant components.

Some models represent the Plant separately from the software, for the purpose of system requirements verification. The Plant model has to show the connection between the requirements of the software interface (motors and sensors) and the overall system requirements that refer to the Plant behaviour.

One of the problems in representing the Plant is that the modeller has to extract the relevant information and knowledge about the Plant. There is no automated process that selects the causally related events and phenomena that relate the software behaviour and the Plant behaviour. Instead, it is left to the insight and experience of the modeller to identify the relevant elements. This is error-prone and difficult to learn.

Multidisciplinary aspect

The first sub-problem is that the modeller has to gather enough knowledge about the Plant from different domain experts. From this knowledge, the modeller then has to distill parts necessary to be described with the model. So, modelling the Plant actually transforms the knowledge and mental models of the domain experts into the software model.

The second sub-problem is about different formalisms used by different domain experts. Software belongs to the discrete domain, thus its descriptions (models) are suited to be described with discrete models, such as automata, discrete message passing, or state machines, to name just a few. On the other hand, most of the Plant processes are continuous. They are designed by different engineering disciplines that focus on mechanical, electrical and other non-software aspects. To address, design, validate, reason about these aspects, different mathematical frameworks, laws and theories are much more suitable than those used for the software models.

For example, the rollers that move paper sheets in an inserter (a machine that automatically folds papers and puts them in envelopes) are designed by mechanical engineers. In the domain of mechanical engineering, these rollers are represented with differential equations containing their size, rotational and tangent speed, and their stiffness. Mechanical engineers calculate the rollers' stiffness and the pressure they apply to the paper so that the paper moves along without slipping. The software modeller then has to decide which of these details are relevant to be represented in the model – is it the rollers speed in relation to the motor's current? is it the rollers' spatial distribution? and so on. There are no obvious answers to these questions. On the top of that, this knowledge has to be described with a discrete software model.

The problem exists even when the processes are discrete or discretised by domain experts, for example by control engineers. Modern control theory is based on discrete mathematics that takes into account aspects of signals that are not relevant to the software (such as stability and observability). The signals

are represented in a mathematical framework suitable to treat these aspects. Consequently, this framework resides in orthogonally different mathematical domains than that of the software (Henzinger & Sifakis 2007).

Moreover, some Plant descriptions are not mathematical – there are diagrams, blue-prints, informal, natural language descriptions of the Plant which parts the modeller also has to incorporate into the software or the Plant model.

"All models are wrong, but some models are useful."[2]

Modelling an embedded system has to bridge between the physical world and the mathematical (computational) world. In formal verification, the goal is to find out whether the system satisfies the requirement we are interested in. Therefore the model has to mimic all the system components and aspects relevant to the requirement of interest. The requirement is encoded into an expression about a model property. In other modelling techniques, the property is not necessarily checked automatically, so we do not have a formal proof for any statement about the model.

In all these cases, we do not have a formal proof that the model is adequate. A model is a representation of a system, an abstraction that mimics a set of system aspects, processes and parts. A model simulates, imitates an aspect of interest and, as (Simon 1996) explained, we can look at one level of abstraction and learn about the system by analysing the model, while not knowing much about other levels and representing them with rough approximations.

As the quote in the title of this subsection suggests, all models are wrong, but this does not matter, as long as the conclusion we draw about the system, by analysing the model is correct. The problem here is - how do we show that the model is adequate, that, even though it is wrong, represents all the relevant system aspects. How do we know that the modeller did not miss something important or over-approximated some of the phenomena of the system?

The answer is that we do not and cannot have a formal proof for the model's adequacy. One approach is to test the model and show that its behaviour matches the system behaviour. This is not a formal proof of the model's adequacy, but it raises confidence in it. Another approach is to make an informal or semi-formal argument that demonstrates the model's adequacy. Surely the significant part of this argument is the rationale of modelling decisions and steps. In the absence of a formal proof, showing how the model is constructed is relevant in the model's adequacy argument.

2.5 Summary

We defined Mechatronic systems as systems consisting of the Plant and the software-implemented Controller. We explained what kinds of abstractions are made when describing the system. We proceeded to explain the problem frames

[2] Even though the original quote of George E.P. Box (Box 1979) refers to statistical models, this holds in general for models (every model is built for a certain purpose)

approach that explicitly shows the requirements addressing the problem-domain concepts of the Plant and the system's environment. The software implemented controller belongs to the solution domain.

We then listed the modelling techniques that are in the scope of this thesis. They all use state machines or automata based formalisms. As the last step in this chapter, we outlined the challenges when modelling the software-implemented Controller.

Chapter 3

Research design

At the end of Chapter 1 we formulated the thesis' main research questions and design problems. Before we present the answers and propose the solution in the following chapters, we will first describe the research methods we used to come to these answers and solutions.

3.1 Methodological structure of the thesis

Figure 3.1 shows the methodological structure of the thesis. Our top problem is a design problem, and the research followed the design cycle (Wieringa 2014). The arrows represent roughly the steps that fall into the phases of problem investigation, solution design and solution validation. This picture shows the steps connected with arrows that idealise the process into a sequential one. The parallel lines decompose a step on the left side into steps within the block on the right side.

In the following sections, we elaborate on research methods we used in each phase.

3.2 Problem investigation

We started our research with the observation that most of the existing modelling languages and tools do not focus on non-formal modelling steps, which belong more to a methodological aspect of modelling; as a result, the quality of a model depends heavily on the talent and experience of its designer. (In Chapter 4, we talk more about the methods that analyse non-formal steps and how our work relates to theirs.)

Before doing something about it, we first needed to understand the problem fully. Our initial investigation aimed at answering the following questions: "What exactly *are* the non-formal elements?", "Which of them are recognised and treated by other methods, and which not?", "Why are the non-formal elements not treated by some methods?" Finding the answers and refining the

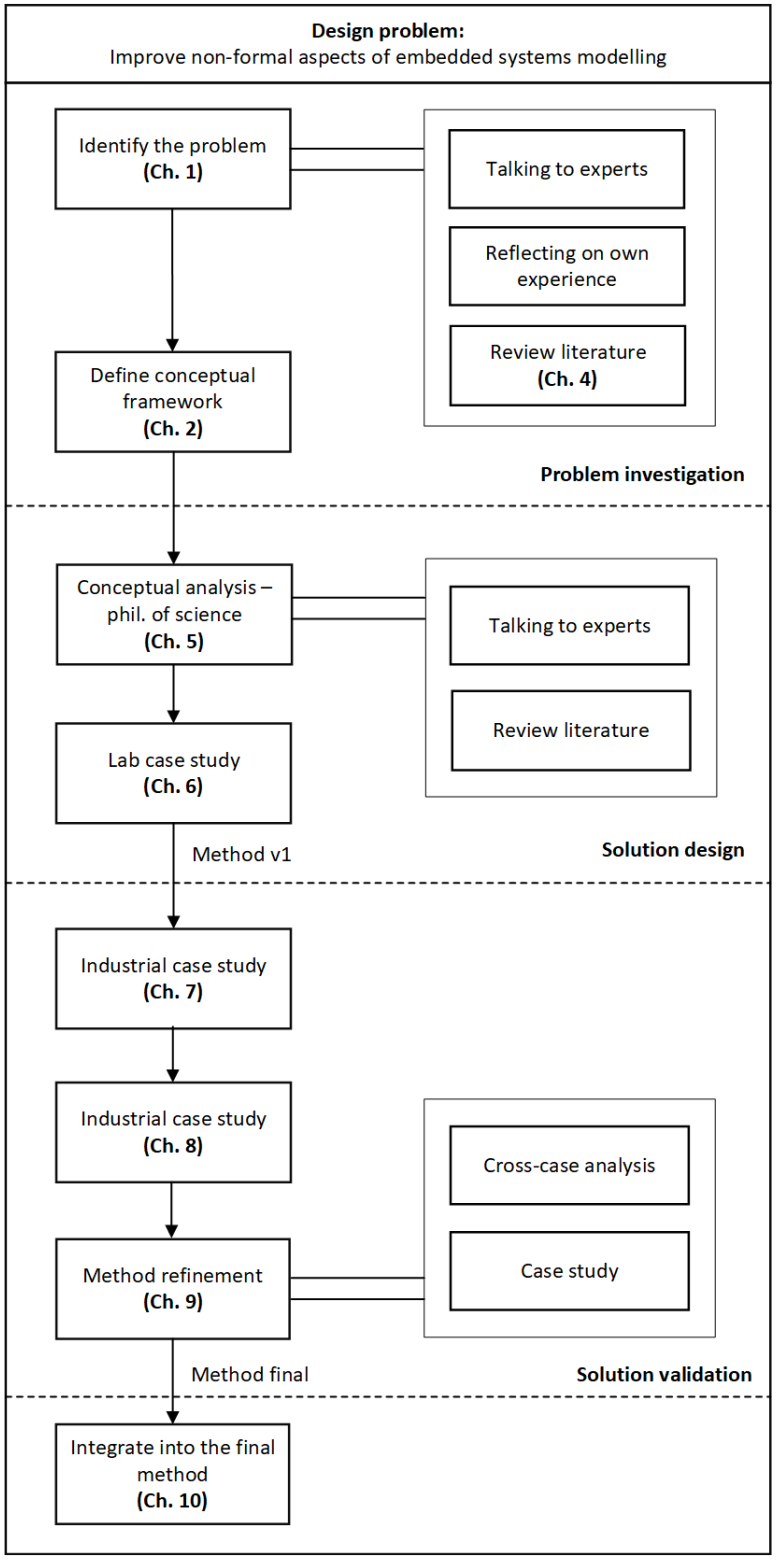


Figure 3.1: Methodological structure of this thesis.

problem into sub-problems facilitated the solution design. In this initial phase, we performed a literature survey, talked to experts, and started working on the modelling case in our lab.

The purpose of the literature survey in this phase was to explore whether researchers and practitioners already address explicitly non-formal aspects of modelling. Our aim was to understand the following: (1) how other researchers analyse the modelling process and its non-formal elements; (2) how other modelling methods treat these elements; and (3) what elements are not addressed. Answering these questions helped clarify the problem and narrow down our questions and goals. We looked at how different software and system engineering sub-disciplines answer the questions we listed above and how they guide the modelling process.

We covered two gaps in the results from the literature survey by conducting a case study. Among research papers on modelling methods, we found those presenting small, hypothetical examples good enough to explain the theory. Another group of papers report on application of the methods on more complex industrial systems. These reports focus on the solution and formal steps rather than on the (non-formal) process of the modellers when designing a model. Given that it is the *process* we want to improve, not the formalism itself, we designed an example / case study in our lab. We designed a verification model of a small mechatronic system, while focusing on the problem area of our interest. Before modelling, we defined the research questions and hypotheses we wanted to test.

Finally, the third method of gaining knowledge about the problem was to talk to the experts in the area, as they sometimes provided insights not described in the literature. For example, formal methods experts recognise the importance of addressing non-formal steps and the importance of a good design, but these "soft skills" are not part of their research. So, they do not write about these insights in their research papers. Interviews, informal talks and reviews of mailing lists uncovered some of the non-formal steps and problems.

We talked to both researchers and practitioners. The researchers provided information on the related work and critically reviewed (an initial version of) our method. Besides, they shared their insights on non-formal aspects of modelling. The practitioners were valuable sources of information on the state of practice. They exposed the problems in modelling complex systems in complicated business and organisational contexts and corrected some of our initial hypotheses on what needs to be improved. The third group of people were experts engaged both in academia and industry. From this exclusive position, as there are few such experts, they communicated their knowledge about how academic methods are used in practice and what research questions are needed to be answered in order to solve practical problems.

3.2.1 Knowledge problems and practical problems

The initial exploration of the problem helped us to understand the problem better. To put it in general terms, it helped us answer the question, "What

is the problem?". The answers led us to refine the main design problem into four sub-problems, and as soon as we started working on them, new questions appeared. These ranged from a meta-problem question: "How are we going to design the method that will solve these sub-problems?" to content-related questions about the nature of these four aspects.

This process—building an artefact to solve a problem and, at the same time, increasing the knowledge of the problem area is inherent in *design science* (Simon 1996). As Hevner et al. (2004) observed, doing design science means problem-solving and answering research questions.

To structure the research questions and design problems, we adopted Wieringa's (2009) nested problem-solving framework. This framework distinguishes practical and knowledge problems as part of the design science process. As the framework suggests, to solve practical and knowledge problems we need different problem-solving methodologies; structuring the research this way guides choosing the appropriate research method for these problems.

Given that the high-level problem is practical, the design process followed iterative cycles of problem investigation, solution design, solution implementation and solution validation. The table in Fig. 3.2 shows our activities within the main phases of the cycle.

3.3 Method design

Our lab case study was complex enough to provide valuable insights on non-formal modelling steps and validate the first tries in designing our method. At the same time, it was simple enough and did not require expertise we did not have or significant amount of time. We designed models, made an inventory of modelling steps, reflected on them and formulated them into modelling process steps. We presented the results to colleagues and received their critical feedback.

In parallel, based on the insights we gathered in the problem analysis, specially on those found in the philosophy of science literature, we continued with conceptual analysis. After reading selected papers on models in the problem investigation phase, we continued with the following steps: (1) organising two workshops with philosophers of science in which models in (software) engineering and verification were discussed (2) Designing a taxonomy of modelling steps; (3) performing additional literature study.

This phase resulted in the initial version of the modelling method. We continued by implement, validate and refine the method in the next phase.

3.4 Method validation and refinement

We used our method in three cases. We performed the following steps: (1) case 1: action research in a company; (2) case 3: action research in another company; (3) cross-case analysis; and (4) case which was a part of a research project.

A = action problem
D = design problem
K = knowledge question
C = conceptual analysis

- A Make the modelling process systematic
  - A Investigate the problem
    - K What are the existing methods?
    - C What are the non-formal modelling aspects?
  - D Design a modelling process
    - A Assemble/define your own
  - K Validate
  - A Use it
    - Acquire action cases
    - Perform the action cases
  - K Evaluate
    - D how to evaluate
      - A Reflect on use
      - A Discuss with software engineers (focus group)
- A Improve confidence in the model
  - C What is "confidence"?
  - Justified true belief that the model is adequate
    - K What constitutes the justification argument?
    - D How to gather all the elements of the justification argument?

Figure 3.2: Knowledge and design problems within the research with the goal to improve modelling process. The goal is divided into two actions.

3.4.1 Action research

Action research brings a researcher and a practitioner close together. Most of the action research advocates in software engineering come from the information system research community. The reason for this is that the action research, together with other qualitative research methods, has the value in explaining what goes on in organisations (Avison et al. 1999). As such, it is "widely cited as suited to the study of technology in its human context" (Baskerville & Wood-Harper 1996).

In other areas of software and system engineering, action research is used under different names. Wieringa (2014) addresses it as "technical action research" (TAR) to emphasise that it is an action research driven by an artefact being designed to address problems indicated by the client. In our research, the treatment, the designed artefact, according to this definition, is our modelling method.

Potts (1993) suggests the "industry as laboratory" paradigm. He points out that the problem with applied research in software engineering is that researchers start from a problem they observed in practice and work on the solution in isolation from the practice; in the meantime, the problem in practice often changes, and the final product is the solution to a non-existing problem. Also, many researchers use the term "case study" to describe qualitative research purely observational in its nature as well as the research done within which not only the problem is explored, but also something is being done about it.

Even though we only remotely looked at organisational aspects that influence modelling, the modelling style and the steps that are the results of the modeller's technical skills in fact, are the human aspects of the application of mathematics in system verification. Also, given its informal nature, the modelling process can be considered an organisational aspect.

After doing the first made-in-lab case study, we carefully chose the next cases. The first requirement was finding a more complex modelling problem with multiple domain stakeholders and experts. The second requirement was to have a system of interest similar to the first case, a mechatronic system. The third requirement was to have the modelling problems embedded in similar organisational contexts as they are also important in defining the limits of the generality of conclusions (Kitchenham et al. 2002). We prepared the case study protocol following (Yin 2003.) and followed the principles suggested by Davison et al. (2004). They are as follows: (1) define well in advance the relationship with a client; (2) follow the cyclical process model; (3) if possible, clearly articulate a theoretical framework on the phenomenon of interest; (4) make sure a change through action is achieved; and (5) reflect on the actions from both research and practice perspective, dividing facts from judgements.)

We performed the first industrial case study in a company that made systems similar to our initial toy-example, but much more complex. We validated the steps for model construction and model justification and continued with the next case. In this next case we focused on model justification. Following was

the cross-case analysis and a case in which we additionally built-up the part of the method on collecting modelling assumptions.

In both of these industrial case studies we looked at event-based software layer, abstracting away the part that interfaces the domain of control engineers. In the last exploratory case study we looked at the interaction with control engineers.

Criteria for the case selection To summarise the choice of our cases, the criteria was, after having a small-scale, in-lab case and validating the hypothesis with our colleagues, to have a more realistic case. These are our criteria.

- An industrial case of mechatronic system.
- Model-based software development is used in the organisation.
- There is a real, concrete problem and part of this problem is system verification using model-based methods and techniques.
- There are users of the model that can provide the domain knowledge and have interest in the modelling result.
- There are stakeholders of the model that can provide feedback and an answer to our validation questions about our model itself.
- The modelling problem is small enough so that we can complete it in a given time, but it is at the same time complex enough to provide insights into the practical use of our method.

3.4.2 Validation

As already mentioned, we validated the initial version of our method by showing it to colleagues and other researchers. We then used the method in two industrial case studies and performed technical action research.

Wieringa (2014) distinguishes a three-level structure of technical action research. At the same time, the researchers themselves play three roles. They are as follows.

- A helper - the researcher helps solve a client's problem in the case.
- An empirical researcher - the researcher answers some validation knowledge questions about the treatment/artefact they designed.
- A technical researcher - the researcher designs a treatment / artefact intended to solve a class of problems.

Similarly, Muller (2020) emphasises the different focus of industrial practitioners and scientific researchers in systems architecting, and names their activities meta^0 (product design), meta^1 (applying an architecting method), meta^2 (designing an architecting method) and meta^3 (designing a method to research architecting methods).

Researcher's role: helper	Researcher's role: empirical researcher	Researcher's role: technical researcher
Design a model for the purpose X that contributes to solving client's problem P, in the context C	*Apply / design a method to construct and justify a model for the problem and context of specific case*	*Generalize the method for a classes of problems*

Figure 3.3: Different role and research and design activities in technical action research applied in the case studies.

In Figure 3.3 we apply the division to roles described by Wieringa (2014) and position our activities in the industrial case studies we performed.

To validate the models we made, we either made them together with the experts and reviewed them regularly or made them ourselves and reviewed them regularly (plus we applied our method for model justification steps).

In the cases, we applied our method, and by working in iterations and communicating with the domain experts and practitioners, we observed which steps were useful and which needed adjustments. In some cases, we performed informal or semi-structured interviews.

Finally, we generalised findings of the cases to similar systems and modelling problems.

In social sciences, opinions about a case study generalisability are different (Gomm et al. 2000.), from those that claim that single cases are not generalisable to those who introduce the concept of "fittingness" and "similarity" that other researchers can use for estimating how a described case can be used in their own contexts.

There is a group of social scientists (Yin 2003.) and software engineering researchers (Kitchenham et al. 1995), (Lee & Baskerville 2003) who identify criteria for a proper conduct of case studies so that some general findings can be conceived. Besides, we conducted more than one case study and performed a cross-case analysis.

As described in (Wieringa 2014), "after explanations for case observations have been found, we assess the generalisability of these explanations by assessing in which architecturally similar cases similar mechanisms could produce similar phenomena." In other words, we generalise by using analogies. The analogic inference we make to generalise to similar cases is based on architectural similarity.

Chapter 4

Related work

4.1 Introduction to related work

As we already explained in the motivation for this research in Chapter 1, most of the modelling methods for software verification focus on formal aspects, such as formalisms, languages and tools, and much less on the steps of modelling process. We, on the other hand, focus solely on non-formal aspects of modelling, that is the modelling *process*. We juxtapose the two focus areas in Figure 4.1, which we will elaborate in Chapter 5.

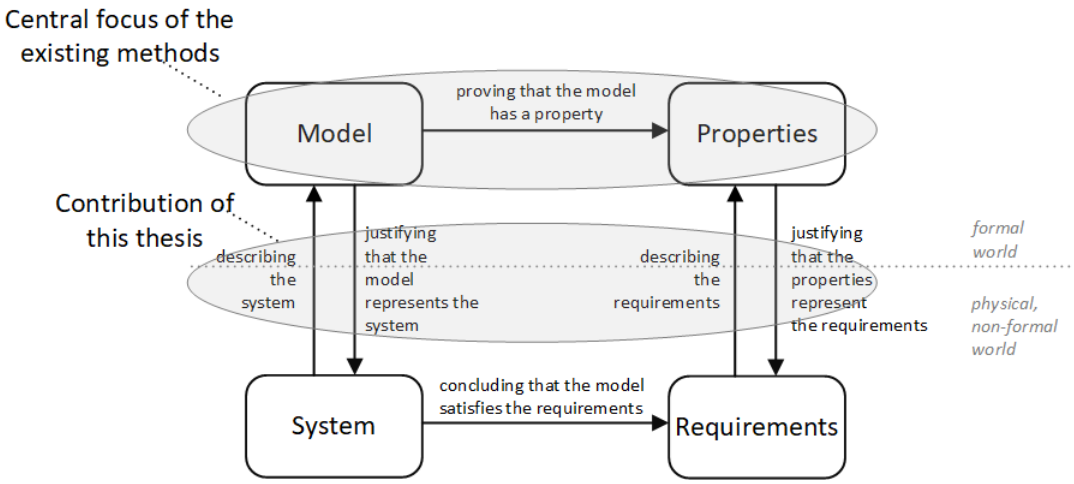


Figure 4.1: The main focus of the existing methods and the focus of this thesis.

In this Chapter we address the following research (knowledge) questions.

KQ1: What are the existing methods and techniques that support non-formal aspects of model **construction**?
KQ2: What are the existing methods and techniques that support non-formal aspects of model **justification**?

Table 4.1: Research (knowledge) questions (KQ) addressed in this chapter.

As explained in Chapter 2, the scope is plant models of cyber-physical systems behaviour, created for software design.

The related work focusing on the modelling process comes from the research work in the following disciplines and sub-disciplines.

- Software engineering approaches
  - Problem oriented methods including problem frames (Jackson 2000) and problem oriented engineering (Hall & Rapanotti 2016)
  - Object-oriented (OO) approaches
  - Model-driven design (MDD) and related approaches
  - Model-based systems engineering (MBSE)
  - Conceptual modelling
  - Combined formal and non-formal techniques
  - Other approaches including requirements engineering for software design
- Systems architecting
- Philosophy of science

In the following sections we give a brief overview of how the researches address non-formal aspects and how our approach relates to theirs.

4.2 Problem-oriented approaches

Problem-oriented approaches go beyond software and explicitly address the overall (problem) context that requires a (software) solution. The approaches build on the problem frames technique of Jackson (2000).

4.2.1 Problem frames

In their seminal paper, Zave & Jackson (1997) explicitly addressed the role of domain knowledge in software design. Jackson developed the problem frames technique (Jackson 2000), in which he proposes to decompose the problem before designing the solution. He also proposes a number of classes of general problems named *problem frames*. For them, a (general structure of a) solution is known.

Each problem frame can be described with a specific *problem diagram*. Also, Jackson names *frame concerns* that are not explicitly described with problem diagrams, but are essential to address to ensure a correct software solution.

Concerning our method, this is a valuable technique to represent the modelling decisions that lead to a model, including the domain decomposition and relating the plant and the requirement.

4.2.2 Extensions of problem frames

Based on the problem frames approach, a number of researchers proposed its extensions as well as formalisation of some of its steps.

R. Seater and D. Jackson (Seater et al. 2007) proposed a software specification technique, starting from the requirements. Instead of formally proving that the composition of the plant and the machine satisfies the requirement, they move from the initial requirement toward the software requirement by transforming stepwise until they reach a requirement at the software interface. While doing this, they write down the assumptions (they call them 'breadcrumbs') on the domain and an argument as to why they replace the requirement in the previous step with the one closer to the software interface. The biggest shortcoming of their technique is, as they say, "the burden placed on the analyst to come up with breadcrumbs..."This is where our work is focused - on a systematic collection of the assumptions.

In his recent work, Jackson (2021) addresses the gap between users' understanding of the system and software designers'. To bridge this gap, he defines "concepts" that enable describing users' ideas, their mental models, and the conceptual models that the software designers have in mind when designing the software.

Heisel developed *agendas* (Heisel 1998), "a list of steps to be performed when carrying out some task in the context of the software engineering". Among others, there are agendas for the requirements elicitation, agendas for specification development from requirements, and different agendas for the software specification (Heisel & Souquières 1999). In the agendas, the transition from informal to formal is performed in one of the first steps of the requirements elicitation, whereas we formalise only the last steps when the complete knowledge about the object of modelling is available. The advantage of their approach is that this allows them to formally analyse all the steps after the transformation to formal descriptions, so agendas are equipped with the valuation criteria for most of the steps. The disadvantage of early formalisation is that they have to fix properties and the behaviour at a stage when they still may be vague, which forms another source of modelling faults.

Choppy and Reggio introduced a General Property-oriented Specification Method (Choppy & Reggio 2004) where they write assumptions (they call them properties) in table cells whose both columns and rows are indexed with components found while decomposing. The idea of mapping the assumptions to components is similar to what we do when focusing on the assumptions about

system components. Their framework is restricted to the use of labelled transition systems.

4.2.3 Problem Oriented Engineering (POE)

Hall, Rapanotti and Jackson developed a formal conceptual network based on problem-oriented perspective (Hall, Rapanotti & Jackson 2007) where they described formally (in sequent calculus) the following steps: the identification and clarification of system requirements; the understanding and structuring of the problem world; the structuring and specification of a hardware/software machine that can ensure satisfaction of the requirements in the problem world; and the construction of adequacy arguments that show the system will satisfy its requirements.

Hall & Rapanotti (2008b) focus on the design aspect of (software) specifications and, like us, recognise its creative element in it. Also, like us, they focus on the process of design software artefacts, not only on the outcome of it. They model the process of iterations in problem exploration and solution exploration. They recognise that problem validation, the risk of solving the wrong problem is transferred or shared between the stakeholders. One of the domains the authors applied this approach was a safety-critical domain (Hall, Mannering & Rapanotti 2007) (Mannering 2010)

The work of Hall and Rapanotti in problem oriented engineering evolved into a design theory of software engineering (Hall & Rapanotti 2017). While there are many commonalities in the starting premisses of our work with those of Hall and Rapanotti, one major difference is that we do not formally capture steps such as problem understanding and problem progression. We are on the other hand focused the modelling-specific steps and assumptions and categorising them. In that sense, our approaches potentially complement each other.

4.3 Object-oriented approaches

4.3.1 Object-oriented software engineering

Regarding general methodological support of modelling, object-oriented (OO) techniques have done a lot to explain modelling steps, but their scope is tightly bounded to the framework of object orientation and its concepts.

The OO methodology provides guidelines for dealing with complexity and recognising different structures during classification and abstraction (Booch et al. 2007). Many examples illustrate good modelling practices within OO. Examples and manuals distinguish modelling phases: analysis, system design, and object design; they also provide guidelines for further structuring the process are given (Rumbaugh et al. 1991). These steps are not presented in the form of a cookbook, given that "the design of [complex] systems involves an incremental and iterative process" (Booch et al. 2007). The process itself is supported by the Unified Modelling Language (Booch 2005).

Some of the elements of OO approach can be used as methodological support for methods other than OO method. However, the steps and best practices of OO methodology are focused on the *software*, not the plant modelling.

Design patterns

The OO approach is designed to be reusable. After all, the idea of design patterns in software engineering was conceived as an OO programming method. Patterns document design solutions for recurring problems in software design. A pattern describes a problem and its context, the elements of the solution and the trade-offs between applying different patterns (Gamma et al. 1995). In their collection of common software design problems, Coad et al. (1995) present patterns together with problem-solving strategies. Design patterns are not only solutions that need to be instantiated, they also explicitly represent (a part of) the experts experience.

4.3.2 Domain-driven design (DDD)

Object-oriented methodology focused little, or at least not explicitly, on modelling of the software environment - until Evans (Evans 2003) put it into scope with the technique he named domain-driven design. He starts from the same premise that Jackson starts that the software is changing the surrounding world and that the representation of this surrounding world is not a trivial task. He introduced the term domain (the same as Jackson's problem world) and gave useful patterns (object-oriented) to model the domain. Even though his method is used for implementing OO-based design (Vernon 2013) because it introduces the importance of analysing the domain (the plant and its concepts), parts of it go beyond the OO framework. The domain model and guidelines on how to come to it are shared with model-driven approaches (Brambilla et al. 2017).

4.4 Model-driven software engineering (MDSE)

In Chapter 2, Section 2.3.3 we referred to the book of Brambilla et al. (2017) in which they define different "model-driven" approaches used in the area of model-driven software engineering. These techniques range from MDD, in which the model is the primary artefact of the development process to approaches in which software models are important but not the key artefact. In the area of MDSE, a lot of work has been done to develop and mature techniques and tools. The methods and tools in MDSE typically have semantics and consistency checking, to be able to verify the model. Regarding the model construction, as mentioned in Chapter 1, when DSLs are used, they facilitate communication with domain experts.

Recently, the awareness has been growing that other aspects are also important. Experts voice the importance of understanding modelling process as well and propose to introduce modelling as an independent scientific discipline (Cabot & Vallecillo 2022). In this article, the authors emphasise the

pervasiveness of modelling, the importance of abstractions which is the starting point or our this work. And in relation to the development of Domain Specific Languages, which ease the task of modelling the domain, the authors note that "the first step to evolve toward a proper modeling approach is making the DSLs explicit to clarify the important knowledge of the domain that should be represented and exchanged."

Similarly, Thalheim (2022) notes:"Models and modelling is the fourth dimension of Computer Science!" and envisions modelling as a new sub-discipline to achieve "a systematics development of a model and modelling culture." The results of this thesis aim to contribute to the mentioned systematics.

4.5 Simulation models and conceptual modelling

In the community that designs simulation models, model validation is an integral part of the modelling process (Murray-Smith 2015). For mathematical simulation and computational models, a process is proposed to iteratively test the simulation model, to perform experiments until it reaches desired level of confidence. In this community, one approach is to categorise the uncertainty about the model correctness into the aleatory—inherent and unavoidable and related to the statistical nature of the physical phenomena being modelled; the second one is the epistemic one and comes from a lack of knowledge, and is possible to reduce by iterating the model and gathering necessary additional knowledge. Our method can help in the latter.

In the area of conceptual modelling for simulation models, (Robinson 2008) outlines the need to guide the modelling process and model justification. He defines not only a conceptual model but also conceptual modelling. It consists of defining the model purpose, determining the modelling and general project objectives, identifying the model outputs, identifying the model inputs (experimental factors) and determining the model content, assumptions and simplifications. As we will describe in later chapters, we name these steps as well (except for those specific to simulation and experimental models) and related to simplifications, we look in more detail into the relations between the model and the target and related modelling steps.

We have already mentioned that Thalheim (2022) proposed introducing modelling as a new sub-discipline in computer science. In his work towards a theory of conceptual modelling he names important concepts of both the model and modelling activity(Thalheim 2010) and (Thalheim 2012).

4.6 Combining informal and formal methods

Model checking methods provide tutorials and teach the formalism and the tool on several examples. These examples also illustrate how to model problem domain. Every formalism has fixed syntax and semantics, and we have to use it to describe the plant. Sometimes, when there is no exact element in the language

to describe certain phenomena, the modeller has to find a workaround to use the concepts of the language to describe the physical world. Regarding model validation, model-checking tools often provide simulation tools. By executing the model, a modeller can test if the model behaves as intended.

Several authors recognise the need to combine formal and informal methods "...to make formal methods easier to apply and to make informal methods more rigorous." (Bruel 1998). An example of how engineers can define behaviour using "user-friendly" UML state machine diagrams combined with model-checking techniques is given in (Grobelna 2020).

UML state diagrams are familiar to software engineers but not necessarily to different domain experts in other disciplines. As mentioned in Section 4.4 the idea of domain-specific languages brings closer the domain experts and software engineers(Brambilla et al. 2017). An illustration of transformations of MDD approaches to formal ones is outlined by Chebanyuk & Markov (2016).

4.7 Goal-oriented requirements engineering

In goal-oriented requirements engineering (GORE), requirements are modelled and analysed as goals. Goal models "capture interactions and tradeoffs between requirements"(Horkoff et al. 2019). They are also used in software engineering and conceptual modelling.

Typical and most widely used and spread techniques and mostly developed are KAOS and Tropos goal-oriented methods.

The KAOS (Knowledge Acquisition in autOmated Specification) method (van Lamsweerde 2009) starts from high-level goals and refine them to subgoals that have to be achieved by a composite system. This methodology also views the system as combination of the software and its environment (although here, different terms are used for the plant and its components. This method is designed for requirements elicitation and refinement, so it ends before the phase of formal specification and verification. Goals are identified, refined and formalised, and tools support detecting conflicts between goals. When decomposing goals, a modeller looks not only for subgoals but also to the obstacles for the achievement of the goals. In order to identify the subgoals and obstacles, it is necessary to know the relevant plant properties. Recent developments include developing a modelling language for requirements and architecture decisions analysis (Busari & Letier 2017).

Tropos, i* and related frameworks support software development in early requirements, late requirements, architectural design and detailed design phases (Giorgini et al. 2005). The system here is described through goals to be achieved by it. The i* model describes actors (agents, roles or position.) Recent developments include the integration of multiple frameworks and automated reasoning (Nguyen et al. 2018).

The systematic literature study of Delima et al. (2021) shows that these two methods stood the test of time and are being further developed and extended.

4.7.1 SCR Method

Another method formulated to specify the requirements in the environment and the software is the SCR (Software Cost Reduction) method based on the four variables model of Parnas (Parnas & Madey 1995). The method was developed and formally underpinned by other authors, one of which is Heitmeyer (Heitmeyer et al. 1996). The SCR method was successfully used in some real-life, safety-critical systems.

The four-variable model introduces controlled and monitored quantities in the environment, input and output variables and different mathematical relations describing the requirements, the environment and the system together with the software. In a case study (Heitmeyer & Bharadwaj 2000) Heitmeyer identifies steps necessary to provide specifications for the requirements, system and the software. These steps are not a method for the model design, but part of the SCR method suited for the four variables model.

The literature search did not uncover new research work for this method except for the (Bourguiba & Moa 2012). However, as Mannering (2010) these methods have been integrated or interfaced to other methods, including model-checking methods.

Table 4.2 summarises the findings on what non-formal aspects are addressed in the software modelling methods.

4.8 Modelling in systems engineering, design and architecting

4.8.1 Model Based Systems Engineering (MBSE)

The International Council on Systems Engineering (INCOSE 2023) defines Model-based systems engineering (MBSE) as "the formalised application of modelling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases." (David D. Walden 2015).

In MBSE, the focus is on the *system* models. The most common language used for MBSE is SysML (*SysML - Open Source Specification Project* 2023). It enables describing the following: functionality, structure, state behaviour, flow behaviour, components' interactions and parametric models (Douglass 2016).

By its definition, MBSE supports systems engineering (SE), a discipline with established standards, processes and definitions that guide scoping and decomposing a (modelling) problem. Furthermore, tool vendors provide guidelines for designing models using a specific tool or language and provide illustrative examples. There are guidelines like that of Mhenni et al. (2014) for systematic model design using SysML. All of these are specific to the domain and practice of SE. The primer of Long & Scott (2011) guides defining the scope and purpose of modelling.

Method	Model construction	Model justification
Problem frames	Problem domain description	
POE	Design process formal description	Validation process formal description
Model checking	Examples	Correct by design; Model simulation
OO	Design patterns, process, best practices, examples	Semantic and consistency checking
MDSE	Modelling templates	Semantic and consistency checking
MBSE	Modelling templates	Semantic and consistency checking
Simulation models	Templates, process steps	Experiments with models
SCR models	Modelling steps	Semantic checking
GORE	Modelling of goals	Semantic check

Table 4.2: Support for non-formal aspects in software design - summary.

These methods and guidelines are focused mainly on the systems engineering activities; our focus is one of the views – state-based, and design method for state-based models only; also, we look at more generic modelling concepts unrelated to the concepts of SE.

4.8.2 Systems architecting

Systems architecting has a modelling and design component. The concepts identified in architecting process relate to non-formal aspect of modelling.

Muller (2011) mentions modelling as "one of the most fundamental tools of an architect". The recurring principles or fundamental concepts inherent to architecting are as follows: communication, understanding, providing overview and insight, awareness of unknowns and uncertainties, articulation of goals and means, customer and life-cycle context, being sharp and factual and feedback and iteration. We recognise these principles as relevant and important in designing models of our interest; the articulation of goals and means is also mentioned in our method.

To reconcile the different views and mental models of different stakeholders and the architect, Muller (2014) uses conceptual models. Engen et al. (2022) found conceptual models helpful in exploring the impact of early-phase project

decisions. As we will show in later chapters, we also use formal and informal models for explaining the end-model.

Architecting is also a design activity. Sandee et al. (2006) recognise its iterative nature and provide a process called "threads of reasoning" that can structure problem analysis, scoping the goals and identifying conflicting requirements. They suggest specific techniques to create insight into conflicts and trade-offs the architect will have to make.

4.8.3 Design process

The steps addressed by modelling and approximating software and system design **process**, such as V-model or spiral model, also hold for the model specification design, especially in the area of model-driven software design where the model *is* the designed artefact and the code is being automatically generated.

In systems engineering, (Bonnema 2018) distils the reference model for different approaches to model the system design process; the way of thinking for multidisciplinary system design (Bonnema & Broenink 2016) guides the engineer to find issues not previously identified. Such guidance can be valuable for the modeller in the initial phase of understanding of the system and the problem, with which the modelling starts.

In system architecting area, (Haveman 2015) provides support, a structured process that guides the design of simulation models in the early (conceptual) stages of system design. In the early stages of design, a common problem understanding among the stakeholders yet to be formed, so a method that complements the technical aspects is necessary and enhances the communication between the stakeholders.

4.9 Philosophy of science

The philosophy of science has been addressing the functions of models in empirical sciences (Apostel 1960), the relations between a model and a modelled system (Woods & Rosales 2010), (Laymon 1989), the relations between computer simulations and modelled world (Laymon 1990), the reasons why certain kinds of simplifications are performed (Chakravartty 2001) (Colburn & Shute 2007), and under which conditions (McMullin 1985) (Weisberg 2007).

Smith (1985) brings forward the limits of a correctness proof of computer program correctness. Cartwright (1983) discusses the role of models in physics.

In their work on analysing the role of models in science and engineering (Boon & Knuuttila 2009) and (Knuuttila & Boon 2011) argue that the models are more than just representations of their target systems and propose to consider them as epistemic tools. In their illustration of their account of Carnot's heat engine modelling, based on his writings, they argue that there is epistemic value in model construction. They, therefore, propose not only to analyse models but also the modelling activity. They argue that the "justification of a model is partly built in the process of modelling". The authors especially look

at "engineering sciences" in which, through modelling while designing to achieve specific function, the goal is to understand, predict or optimise the behaviour of devices.

In another article, (Boon 2020) notices that engineering science education "often pays little attention to non-mathematical ways of scientific modelling" and puts forward the method of testing the models. The authors propose a method for (re-) constructing scientific models of phenomena (Boon 2020) consisting of 10 questions to determine the concrete elements of the scientific model systematically.

4.10 Summary of the related work

There is an abundance of modelling formalisms, languages and tools designed. Some of them are more academic and not used in practice; the practitioners adopt some of them. Books, tutorials and articles provide modelling examples.

Researchers work on presenting results that combine mathematical (formal) methods and UML or SysML language. Combining the methods can bridge the gap between domain experts and software engineers.

Modelling patterns For some methodologies, such as object-oriented ones, modelling **patterns** provide reusable, well-designed models or templates. These modelling patterns are based on best practices and are standardised solutions for common problems. This way, the designers in one team or organisation do not need to re-invent the models, which saves time, lowers the maintenance effort of these models, and lowers the cognitive load (Sweller 1988) of understanding different models that represent the same domain or a problem.

Design process In software engineering, POE (Hall & Rapanotti 2017) describes the problem refinement and the process of problem analysis and validation, solution specification and validation. These steps are formally described.

In systems architecting, design steps focused towards designing system architecture are described, and various methods propose guidelines for specific architecting phases.

Modelling process A group of researchers in modelling for simulation and conceptual modelling outlines steps when designing simulation models.

Philosophy of science takes a closer look at how models describe natural phenomena. Authors name and analyse different relations between a modelled target and its model. Boon (2020) analyses the role of models as well as a process of modelling in engineering sciences. The insights and findings from the philosophy of science were a basis for our taxonomy of modelling steps that we will present in Chapter 5.

Chapter 5

Conceptual analysis: modelling method

In this and the following chapter, we present the initial version of our method. The design goals we address are shown in Table 5.1, which shows a subset of questions in Table 1.1.

Design problems and knowledge questions
1.1 Design problem Design a method that guides the **construction** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.
2.1 Design problem Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.

Table 5.1: Research questions addressed in this chapter.

The result of this chapter is as follows

- The conceptual model that shows the relations between the system, the requirements, the model and its properties.

- The conceptual model with different modelling steps as part of the requirements verification.
  (We have introduced one of the models in previous chapters but will repeat it here.)

- The structure of the justification argument that the model is adequate.

- A characterisation (taxonomy) of the model construction steps.
  They are part of the solution for the Design Problem 1.1. At the same time, naming them when explaining the model is part of the model justification (that the model is adequate, i.e. that it represents the system correctly with respect to its purpose) - this is part of the solution for the Design Problem 2.1.

- A process (workflow) to follow when designing the model.
  These are the steps describing the modelling process that can be used as guidance and checklist when designing the model (as part of the solution to Design Problem 1.1) and guidance for model justification (as part of the solution to Design Problem 2.1).

5.1 Background

The conceptual analysis presented in this chapter is based on the following: Reflection on our first case, an initial modelling example; Reflection on the findings in literature; and two workshops with philosophers of science (Boon & Knuuttila 2009) with the topic of verification and engineering models. In the latter, we agreed that modelling guidance is needed and that it deserves more attention in software engineering than it was the case at the moment.

5.2 Two conceptual models

This project started with an existing diagram shown in Figure 5.1, created in this project initiators' informal discussions and inspired by a similar diagram proposed by (Wupper & Meijer 1997). The diagram outlines that the model represents the system with respect to the system requirement. The model properties represent the system requirement. The model and the property are formal concepts; the system and the requirement are not; they are non-formal.

It applies to verification models what Smith (1985) called inherent "limitations to what can be proven about computers and computer programs" ((Hall & Rapanotti 2017) refer to other researchers expressing similar limitations and sum it up as a "formal-non-formal divide").

In the next conceptual model shown in Figure 5.2 the system and its requirement, the model and its properties are shown. Between them, we formulate the activities that the modeller performs. The modeller describes the system with the model and the requirements with the model properties. If the model and

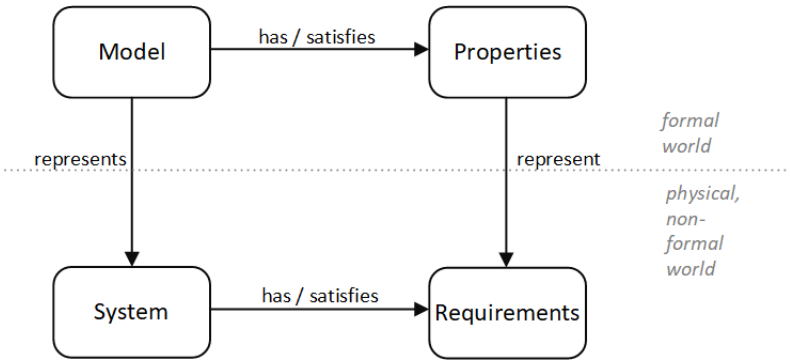


Figure 5.1: Relationships between the model and reality (the system that is modelled.)

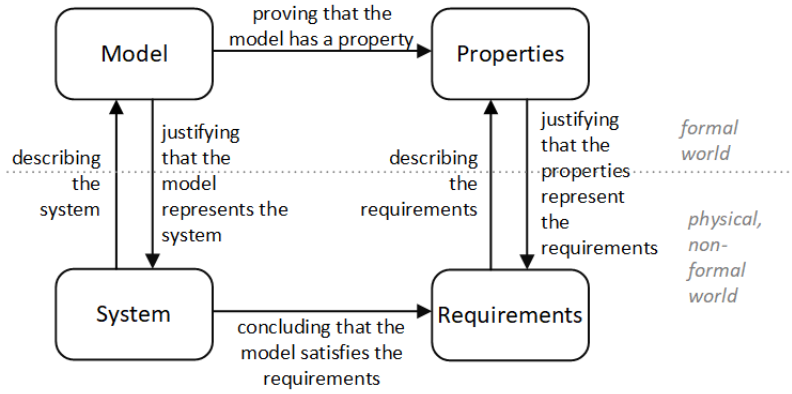


Figure 5.2: Steps performed when modelling.

the property are adequate, then if the model possesses the property, we can conclude that the system satisfies the requirements.

Looking at the design problem 1.1, we ask the (knowledge) question: what are the model construction steps, and can we characterise them irrespectively of the formal verification language or formalism used?

Also, we establish the structure of the justification argument that the model is adequate, which states the following.

(If the Model (M) satisfies Properties (P)) then (the System (S) satisfies Requirements (R)).

5.3 Taxonomy of modelling steps

Compared to other scientific and engineering disciplines, software engineering and computer science are relatively young disciplines. The models that they create do not use the same mathematical frameworks as other disciplines such as for example physics or chemical engineering. Still, the characterisation of the modelling steps in other disciplines is generic and does not relate to a specific mathematical framework or specific formalism.

In this section we show how modelling steps performed in other engineering disciplines and natural sciences characterise also modelling steps when designing the Controller for a mechatronic system. The steps we will discuss are: abstraction, idealisation, approximation, decomposition and localisation, establishing analogies, and establishing causal relationships. Philosophers of science discuss these activities as they are undertaken in science and classical engineering disciplines. Their analysis is part of their search for answers to philosophical questions, such as: "Are scientific theories telling the truth?" and "What does it mean for a model to represent a part of the world?". These questions are looking at the relations between the models and the target they describe which are the same as relations shown in Figure 5.1.

Even though our goals are different – we are designing a method for constructing a system verification model and for justifying its adequacy, whereas philosophers of science ask knowledge questions about the model-target relation, we can use these answers as hypothesis for our more pragmatic question - how do we know that the model represents the system adequately? Our goal is to incorporate the analysis of philosophy of science about the models and their meaning into the method for model construction and justification.

The steps we present we found in the literature that explored mostly physics and fluid mechanics, focusing on comparison between the modelling activities in science and engineering. To the best of our knowledge, a comparable analysis has not been done for embedded systems modelling.

One may ask: If the intellectual process of making a leap from an embedded system to its model is the same process that a physicist undertakes when modelling phenomena in nature; why do we need to analyse these steps exclusively for software engineering? Why don't we just take over the analysis already performed for other disciplines? Our answer is that other engineering disciplines often use directly the models and theories from science and, if necessary, adapt them (Boon 2006). In embedded software engineering, as we explained in the previous section, we cannot directly use control laws, statical equations and other models as parts of software discrete models. Instead, we have to non-formally interpret and combine parts of different models into software engineering models.

Another difference between modelling in other branches of science and engineering and modelling embedded systems is that in science in general, the model is a stepping stone from the observable world to a theory. Science strives for generality and theories apply not only to a single system but to some phenomena in general. Some models in science are idealised in order to explain

an isolated mechanism in nature, but they do not necessarily provide accurate numerical results. A theory that explains a mechanism holds for an idealised model, but when applied to a real phenomena, it may turn not to be completely accurate. (Cartwright 1983)

In embedded system modelling, by contrast, we are not interested in generality, our model has to represent only the system under development. The results we get from verification models are not numbers, they are 'yes' or 'no' answers to questions about the system properties. We cannot afford a wrong answer there, the model has to be adequate, accurate 'enough' to give us the correct answer about the system.

5.3.1 Abstractions, idealisations, approximations - terminology

In the literature of philosophy of science, there is no unique definition of the terms 'idealisation', 'abstraction' and 'approximation'. Some authors use the term 'idealisations' to refer to all kinds of simplifications implemented in a scientific theory or a model. Other authors make a distinction between them and go even further in recognising different kinds of idealisations and approximations.

Before we proceed, we want to make a distinction between a modelling *activity* that we call "abstraction", and a model *component* that we can also call "an abstraction". In this article we will refer to abstraction, idealisation etc. only as to modelling activities.

Figure 5.3 shows the taxonomy of modelling steps.

5.3.2 Abstractions

In our experience the term "abstraction" is one of the most commonly used terms when describing a system model. We define it here and distinguish it from other modelling steps.

In system modelling, abstraction means leaving out, omitting describing some of the system components, properties, processes, or aspects. Underneath every abstraction step lies, often implicit, reasoning why omitting certain information about the system results in a model that still adequately represents the system. Take for example, a verification model of the software embedded in the inserter we mentioned in Sect. 2.4.2. The software controls transport of papers and envelopes from their designated feeders through the machine modules, to the exit. The software also controls movement of numerous mechanical parts that fold a paper, open an envelope, and insert the folded paper into the envelope. But, in the inserting module, the moisturising of the envelope flap with a stroke of a brush is controlled mechanically. The moisturising takes place simultaneously with other software controlled actions without interfering with them in any way. Therefore, the flap moisturising is invisible for the software, and consequently irrelevant for the software model.

Philosophy of science literature lists the reasons why abstractions are performed. One is the relevance, and the example we gave falls into this cate-

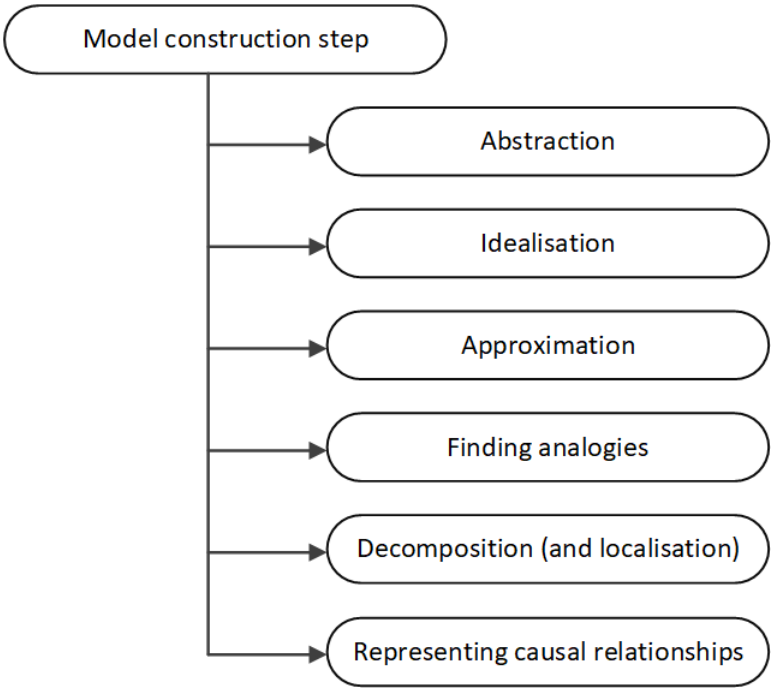


Figure 5.3: Taxonomy of modelling steps.

gory. The property under investigation, the purpose of the model, the desired preciseness of calculation or predictive accuracy do allow to ignore many parameters (Chakravartty 2001) . Another reason is pragmatic, as the number of potentially relevant factors for the property we are interested in, is extremely high, which would make constructing a model impractical (Chakravartty 2001).

In modelling for verification, we start dealing with this complexity in the early phase of the model design, while defining the modelling problem. The modelling problem is decomposed into sub-problems by decomposing the system requirement. Modern mechatronic systems are so complex, that when the requirement asks for a certain functionality, the design problem needs to be decomposed. Modelling all the "sub-requirements" with one model will be a challenging, possibly too complex task for the modeller. Such a model will not be easily comprehensible, and its maintenance would be difficult. Also, many model-checking frameworks ask for being mindful not to design models with state explosion or other problems related to computational complexity. Take for example a requirement for a safe control of a car; it can be decomposed into the following requirements: "When the car is on ice, if the driver hits the brake, it will be overruled by the control", "The cruise control gives the control to the driver as soon as he presses the brake or gas pedal", "The air-bag

is blown if collision is detected". By decomposing the safety requirement into sub-requirements the models are less complex and easier to manage.

There are no firmly established guidelines on what should the 'filter of abstraction' capture and what simply does not count (Woods & Rosales 2010). An analysis (Woods & Rosales 2010) of criteria for abstracting away aspects, parts, phenomena of the modelled target, found the following criteria in the contemporary literature.

- Relevance vs. irrelevance (this one is about what the purpose of the model is)
- Simplicity vs. complexity
- Manageability vs. unmanageability
- Tractability vs. intractability, and the author adds his own, related to the cognitive value of the model,
- Information vs. noise.

The same analysis also categorises the abstractions done in science to those that (1) omit relevant factors for the sake of getting to partial truths that are in some way useful (Cartwright 1983); (2) the abstractions that omit only irrelevant factors, so even if they were incorporated in the model, they would not change the modelling result; (3) the abstractions made for the sake of simplicity, aimed at reaching high cognitive value, rather than providing calculation or prediction. The latter criterion is implicitly present in best practices of modelling in software engineering, where for example layout of diagrams has to be 'clean', readable, not too complex. At the same time, the verification models we use have to be precise, so this last reason should not be taken into account when designing a formal verification model.

In embedded system formal verification, we want the models that give truthful predictions, confirmations and disputes. Does that mean that we cannot afford the abstractions that, when put into the model, would change verification results? By decomposing the problem into computationally tractable and manageable problems, we eliminate partially the problem of having to abstract away for pragmatic reasons. However, there are many modelling assumptions that are not part of the model. These assumptions are critical for the model's adequacy because, if they are not fulfilled, the verification result may not hold. For example, we can say our model describes the system that operates on temperatures between -10 and +50C, but nowhere in our model we represent temperature values.

We do not agree with (Colburn & Shute 2007) that the abstractions we make are made only for the goal of hiding information. The authors there give examples of encapsulation, where higher-level description hide the information about lower-level ones. Specification in higher-level programming language hides information about bits and bytes and so on. This is true for abstractions made

when focusing on abstraction *layers*. We can abstract away control laws implementations, or software drivers behaviour, these are all examples of abstraction layers.

But, modellers also completely omit certain aspects of the system, like in the cases in which they abstract away the plant description and prove only the requirements at the software interface. They implicitly assume that the behaviour on the software interface will result in the desired behaviour of the whole system. The plant components are not irrelevant, but they are (or should be) addressed separately from the model.

When describing the plant, the modeller chooses what aspects to describe and what to leave out. The plant is described by different domain experts, such as mechanical and electrical engineers. For software verification only some of the descriptions given by different experts is relevant. For a mechanical engineer, the stiffness of a material and other mechanical properties are relevant, the dimensions play a role, and their statics. The modeller of the plant for the purpose of software verification needs to know the spatial distribution of some of the elements, how the mechanical elements are connected, and how movement of one influences the movement of the other components.

5.3.3 Idealisations

Unlike abstractions that omit the truth by not describing parts of the system, idealisations introduce distortions.The modeller describes certain system aspects as if they were different than they are. "Idealisations change properties of objects, perfecting them in certain ways, not all of which even approximate to reality." (Woods & Rosales 2010). Point masses in calculating positions of planets rotating around the sun is a paradigm example of an idealisation in physics. A point mass does not exist in the physical world, but in the point-mass models of Newton and Kepler theories they are used to help understand the movement of planets and calculate their positions.

In software modelling, we often idealise events as instantaneous, like opening and closing a valve, for example. Another example is representation of sensors as dimensionless points in the system. Rollers that move papers along a printer are made of soft rubber. On their contact with paper they flatten temporarily which changes their round shape and tangent speed with which paper is moved. However, for some calculations they can be idealised to perfectly round-shaped rollers without compromising the overall result.

Cartwright (Cartwright 1983) recognises two different types of idealisations. The first describe the phenomena in a way that can never be achieved in reality, like for example point masses in physics. These are useful to explain and understand the phenomena, but they never give accurate predictions. The second type are idealisations that can be approximated in reality. For example, if we assume vacuum for the law that calculates pendulum angle and velocity, we could have this in reality, by putting the pendulum in a very low pressure room. The theories based on these idealisations approximate the truth. The second type of idealisations allows for de-idealization (McMullin 1985), which

is relaxing ideal conditions and bringing them closer and closer to reality. In science, they are performed with the goal to simplify theories and make them computationally tractable (Weisberg 2007). Verification models are designed to provide accurate result, and there a 'second chance' to de-idealize is not given.

A special type of the truth distortion is assumption of a worst-case scenario. The modeller assumes extreme conditions, such as extreme temperatures or pressures. These are at the same time the strongest mathematical conditions on the model, and if the property holds under these conditions, it will guaranteed holds in less strong conditions.

To be able to examine the property of interest, the modeller has to invent elements that do not exist in the system, but are necessary to observe what happens in the model.

5.3.4 Approximations

Approximation is assigning a numerical value that is not the exact one, but within an acceptable error boundary. If for example the prediction we want to make with the model is of an order of minutes or hours, than we may choose to round all the values that are below seconds. For example, in model-checking language Uppaal (2023), time passage is represented with time units which have to be normalised to the lowest common denominator of all the time values. If all the processes take time in order of minutes, than one time unit is interpreted as one minute. But if there is a process that lasts order of seconds, all the times have to be converted into seconds, and in the model described with time units that represent seconds. This increases the state space, and complexity of the model. So, it may be convenient, if the property of interest allows, to approximate the duration of that one single process to zero.

Laymon (Laymon 1990) defines a special kind of idealisation, which is a transformational **approximation**. It is a substitution of a term or an expression in a mathematical expression that represents a theory or a law, with another term or an expression, under the assumption that the overall function will stay stable. Transformational approximations do not apply to discrete, state-based models we are focusing on. But, there are other transformations that the modeller invents to have a model that is simpler, smaller, easier to manage, easier to comprehend. For example, the structure of the model maybe a structure that is not present in the system, but that shares the relevant properties with the system.

5.3.5 Abstractions and idealisations relationship

In many cases both idealisations and abstractions are performed at the same time, or even more, they are closely linked. We mentioned an example of dimensionless sensors in the model, but it is difficult to say whether we just abstracted away the dimension or idealised sensors in form of points.

When explaining the model, we do not necessarily separate abstractions, idealisations and approximations from each other. A modelling decision the

modeller finds relevant to address explicitly can be a mixture of these. Still, by knowing the distinction between these decisions, we then have three ways to justify a difference between the model and the modelled system. In case of an abstraction, we left out the elements that do not influence the property we want to prove (or we address them separately). In an idealisation, we added things to make computations in the model easier, without compromising the results interpretation back in the modelled system. In an approximation, we argue that the model is a good-enough approximation of what is the case in the real-world; it will produce a slightly wrong prediction but that prediction is still good enough for our purposes, e.g. within safety margins.

5.3.6 Finding analogies

Analogies represent one phenomenon with another phenomenon of a different nature. In physics, a typical example of analogical model is a model of light based on an analogy to water or sound wave. Analogies are based on similarities of different phenomena.

An analogical model incorporated into a computer-based system possesses phenomena and properties which have no counterpart in the subject (Jackson 2009).

5.3.7 Decomposition (and localisation)

Decomposition is, together with abstraction, often and commonly used term when modelling software. Decomposition is a way to deal with complexity. Bechtel et al. (Bechtel & Richardson 2010) distinguish between decomposition – giving structure to observed system or phenomena, and localisation – tracing what part is responsible for a certain function. They define decomposition and localisation as strategies for discovering mechanisms for articulating structure of the phenomena.

We look at decomposition and localisation in Bechtel's interpretation under the umbrella term of decomposition. Decomposition as we define it, consists of a top-down and a bottom-up type of process. In the bottom-up process, the modeller assigns the structure to the system by recognising system parts that have the same characteristics. In the top-down process the modeller isolates different components and discovers what each one does. After performing a combination of these two types of processes, the modeller is able to show different structures and parts responsible for different functions. In system engineering, well known activities are functional and system decomposition.

One might argue that man-made, engineered artefacts reveal their parts which makes their decomposition easier than the decomposition of phenomena in the nature. However, today's computer-controlled systems are so complex, that there are many possible structures that the modeller has to explore. These systems do not have a single unique decomposition.

For verification and simulation models, the modeller has to perform the following decompositions.

- Decomposition of the modelling problem. The problem is typically expressed in vague terms, and even if it's not it may happen that the requirement has to be decomposed into sub-requirements each of which can be analysed with a different model.
- Decomposition of the system - physical and functional decomposition. The system is built by different engineers, it has different components, functions, performs different processes – all these have to be explored in order first to understand the system and the modelling problem and then in order to represent them in the model. These come from the systems engineering methods.
- Decomposition of the model. Related to different structures of the system are different perspectives from which we can view the system. They usually come from different engineering disciplines responsible for different system functions.
- Level of abstraction with which software and the plant are represented. Even though this is about abstractions, it can be argued that how we establish hierarchies is decomposition, too.

A decomposition of the model and a decomposition of the system do not necessarily coincide. If they do, we have isomorphism between the system and the model components. If not, we argue implicitly or explicitly that the two structures are equivalent regarding the property and behaviour of interest.

We may have a partial isomorphism - some of the model components are isomorphic to the system components and some are not. Or, the high-level model structure correspond to one of the structures we identified in the system, but as we further decompose, it may happen that there is no isomorphism. In systems engineering we talk about mapping of physical components to their functions; and we can talk about mapping of the model components to system functions and/or system components.

The structure of the model depends often on the structure of the system, but it also depends on the modelling method chosen. If we chose an object-oriented method, we will 'see' the system as composition of objects, which will be represented in the model, together with their internal structure, functions they perform, etc.

Finally, hierarchies in system modelling define how detailed the description has to be. For example, we decide if a process is detected by the software without representing the sensor explicitly. The notion of hierarchies is coming from the system and software engineering world. In software engineering, one of the common architectural patterns is to create layers. On one layer the concepts are more abstract than on the layer below. The layer below is an implementation of the layer above it.

Together with decomposition there is a formal or non-formal, implicit or explicit argument of recomposition.

5.3.8 Representing causal relationships

Causal relationships show how the behaviour on the software interface causes the plant as a whole to behave as required. The software can interact directly only with actuators and sensors, whereas the system requirement refers to the plant parts that are usually on some distance from the sensors and actuators. Therefore, the task of the modeller, and of the software designer, is to establish causal relationships between the system components that result in the overall system behaviour.

For example, the relationship between warming up food in the microwave oven, and the sensors and actuators of the embedded software in the oven is: the motor rotates the plate holder, the magnetic element emits the energy when the software sends the signal to the switch, the sensor senses the temperature, and the timer measures the time. Usually this reasoning is performed informally and stays implicit. (Seater et al. 2007) explored how these relationships can be made explicit.

When we model the system as a whole, we do not necessarily separate the software and the plant, so these relationships are formally described with the model. When describing software only, there is, within the software specification, a representation of (part of) the plant and causal relationships (This is what Jackson calls a model of the environment within the software (Jackson 2000)). Variables or structures keep track of sensors' values and the order of events and based on this data they determine the state of the components that are not directly connected to the software.

In the world of system and software engineering we talk about system processes, where performing one step is a precondition for another one. So in this step we also have overlapping with system and software engineering concepts.

While decompositions and components that are the result of abstraction and idealisation focus either on static picture of the system, or if they describe dynamic aspects like processes, they show then their static representation in the model. Causal relationships cannot be pinpointed directly in the model, they are not documented as such, but can be shown by analysing multiple components of the model and their meaning.

To summarise, we list here the following steps recognised in the literature of philosophy of science when designing models for physics and other scientific disciplines as well as mature engineering disciplines: abstraction, idealisation, approximation, decomposition and localisation, establishing analogies, and establishing causal relationships. We describe how they are performed in designing verification models for mechatronic systems software controller.

These steps answering the knowledge question what are the non-formal steps made when describing/modelling the system and the requirement using a formalism, thus making a step from non-formal to formal world.

This knowledge can also be used in a prescriptive manner to explain how the model represents the system.

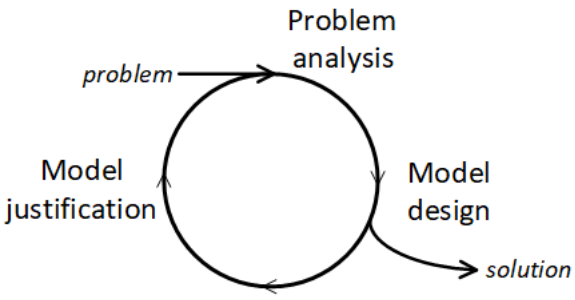


Figure 5.4: Modelling as a design process and as a problem solving activity.

5.4 The modelling process

The steps we identified in the previous section are those that the modeller performs *after* making the decision what parts of the system to represent in the model. These steps show *how* the selected elements are represented in the model. But, these steps do not explain why a system element is represented in the model. Also, they do not explain why it is represented with the given accuracy and not more or less accurate or detailed. We will identify the rest of the model explanation elements through analysis of modelling as a design process.

Modelling is a design activity. In software engineering, one of the commonly used paradigms to describe software design is that of problem analysis, solution design and solution validation (Dasgupta 1991). Different structures for the software development process, such as the V-model, the incremental model, iterative cycle models, all contain these three aspects as phases of software design. Applied on modelling, these three activities are: modelling problem analysis, model design and validation or justification of the model. The diagram shown on Fig. 5.4 shows them positioned in an engineering / design cycle.

In the rest of this section we will list the steps within each of these three activities. We do not suggest their chronological order, given that our goal is neither to describe evolutionary nor incremental character of the model design process. Also we do not investigate relations between all the processes and steps. When identifying these steps, we focus on the steps of the taxonomy presented earlier in this chapter.

5.4.1 Problem analysis

Like most of the design problems in software engineering, a modelling problem is almost always ill-structured. This is the term used by Simon (Simon 1996) to characterise problems that are not well-defined. Translated to modelling, this means that the modelling problem, the system and the system requirement are often formulated in vague terms. It is the task of the modeller to refine the problem definition.

Figure 5.5 shows our classification of the problems that that the modeller has to solve while designing a model.This classification is based on the analysis of the case studies we performed, and on the analysis of software engineering processes in general (Hall, Rapanotti & Jackson 2007).

Analysing the modelling problem

There is no single "correct" model of a system that can describe the system for different purposes. The model and the purpose for which the model is designed are inseparable and together determine how the model will be designed. A number of modelling decisions, such as the choice of an abstraction, depend on the purpose of the model. Therefore, an explicit statement of the purpose is therefore a prerequisite for both, construction and justification of a model.

In case of verification models, the purpose is determined by the property to prove. For a simulation model, it has to be defined what are the phenomena that will be observed. Identification of all these may not be known in detail beforehand. Also, formalising a property or making precise what is observed is part of the modelling process. Therefore, the statement of the purpose of a model may be constructed incrementally.

There can be a number of limitations posed by stakeholders on the model itself. They can concern the choice of modelling language, the decomposition of the model, the resources available to build the model (in case of the model is implemented with hardware elements) and many others. This, too, influences the modelling decisions and narrows down the solution space.

An explanation of practical constraints clarifies why certain modelling choices prevailed over the other which are as good or which would lead to a more simple, understandable or easier to manage model (or whatever is the quality criteria of a stakeholder's interest).

Understanding how the system works

To be able to determine what to model, the modeller needs to understand how the system works, what the system requirements are and what the stakeholders want with the model. The modeller is not necessarily someone from the organisation that designs the embedded system, therefore not someone who knows the system. Very often, the modeller starts with the system and the modelling problem as black boxes and unveils them gradually, by learning about the system and about the problem.

To be able to understand how the system works, the modeller has access to technical documentation of the system, the system stakeholders, and sometimes the system itself, partially or in total. In an industrial organisation, the stakeholders have their own jargon, so the modeller also learns the 'language' of domain experts, or languages, acknowledging that different domain experts sometimes use the same words for different meanings (and different words for the same meaning).

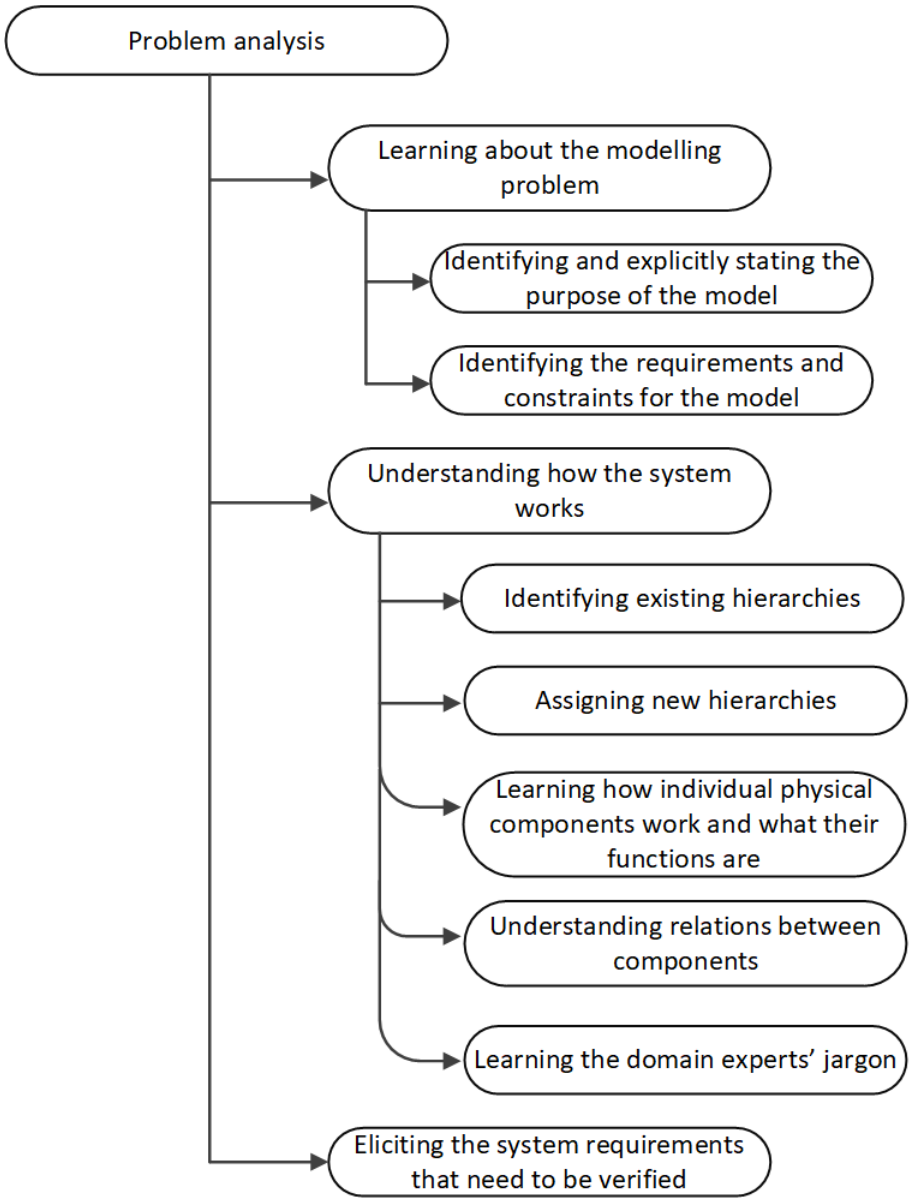


Figure 5.5: Activities that are part of problem analysis during model design.

In this process, the modeller learns about the existing system decompositions. In practice, these decompositions are often mixed, made according to different criteria. Even though they are not 'perfect' they serve purposes of the organisation. But the decomposition is not a tangible artefact, it is a concept, and it is what people assign to complex systems. Therefore, the modeller can also define her own decompositions while learning about the system.

The modeller also focuses on individual components and modules and learns how they work. Of course, it is not possible to get into all the details, but it is necessary to find out how these components contribute to the overall system behaviour. The modeller also needs to establish how different components interact among themselves.

In relation to this process, we identify two additional elements of the model explanation. They are: explanation why a chosen abstraction level is accurate enough and an explanation or justification that the model structure is adequate. In the latter case, it can happen that the structure of the model does not coincide with any of the existing system structures. In this case it might be necessary to justify that the model is equivalent to the model that would have the same structure.

Eliciting system requirements

When modelling is used for design, then the modeller has the role of the software specification designer as well. Inherent to software specification is eliciting system requirements, and this is also done incrementally.

5.4.2 Model design

We distinguish two main aspects of model design (see Fig. 5.6). One is the decision *what* aspects, parts, properties, phenomena etc. of the system will be represented in the model. The other are the concrete modelling steps, and the decisions that have as a direct result a part or a segment or an element of the model constructed. These decisions are about *how* the system will be modelled and we have already talked about them in Sect 5.3, in the taxonomy of modelling steps.

We classify the decision on what to model as part of the solution design, rather than the problem analysis because it is up to an extent a decision of the modeller what will go in the model.

Another aspect of the solution invention is that the modeller does both top-down decomposition of the problems and bottom-up recognition of previously solved problems. Top-down and bottom-up thinking always come together in design. Cognitive science showed that a designer recognises 'templates' - previously solved problems, and in case of modelling the modeller will recognise previously solved problems and will map them to her own 'pattern library' that resides in her long-term memory (Cross 2001). The bigger the experience, the higher number of solutions and choices that the modeller has. Experiments showed that "experts have acquired over the years of experience and training

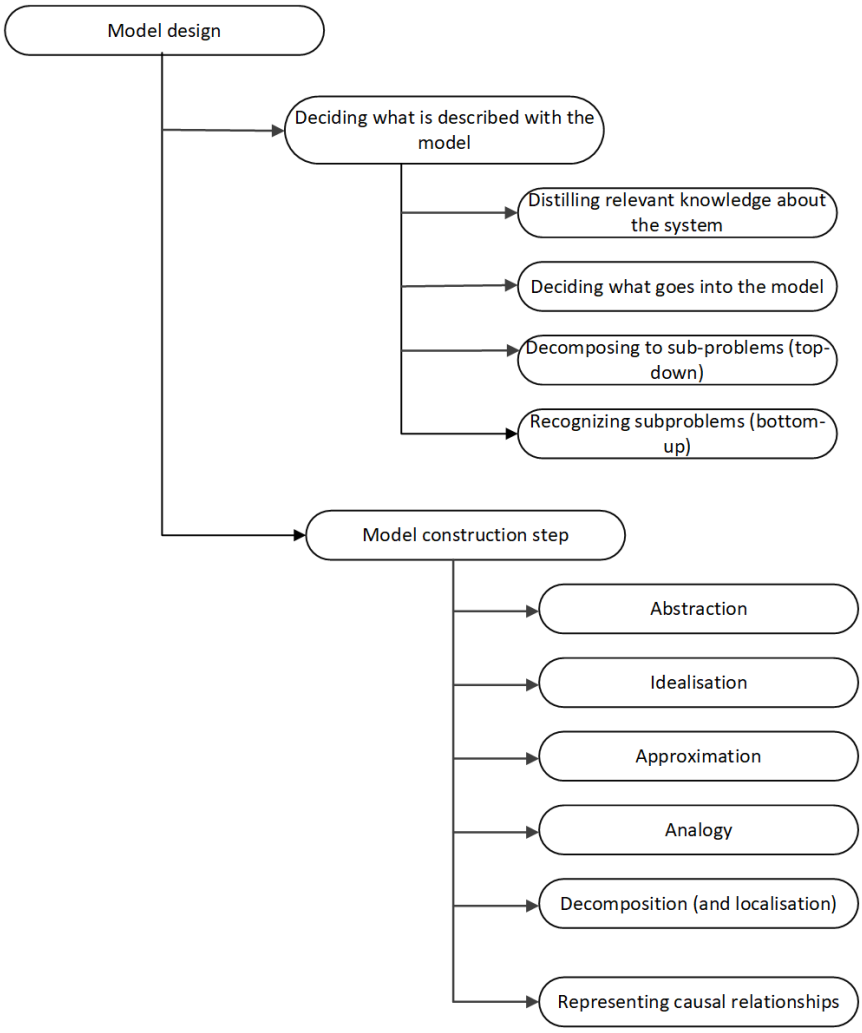


Figure 5.6: Model construction steps and decisions.

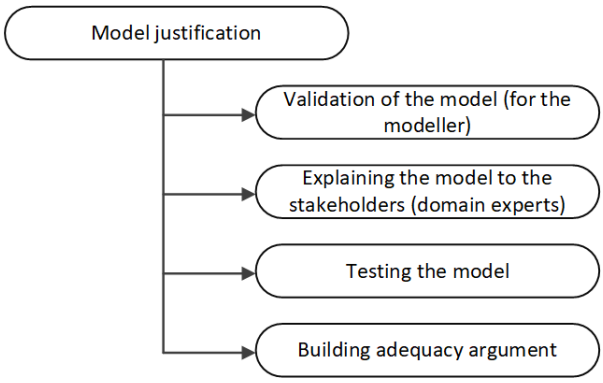


Figure 5.7: Aspects of justification that a model is adequate.

the ability for rapid decisions making and that they outperform novices in many tasks." (Robson et al. 2021).

Among the decisions that concern the model components in an indirect way is the decomposition into the model subcomponents. There is always a mapping between the model and the system architecture, but it is the modeller's decision what structure will be represented in the model. Assigning hierarchies and decomposing is a top-down strategy to deal with complexity.

5.4.3 Model justification

Showing that a model is adequate can be done in a non-formal or semi-formal argument. The argument will, among possible other elements, contain the justification of the modelling steps that we analysed as part of problem analysis and model construction. So, the justification steps will be: justification of the problem decomposition, justification of an abstraction etc. Most often, justification is not structured, but is an implicit part of the communication between the modeller and the stakeholders.

Fig. 5.7 does not show the structure of the justification argument, but it emphasises the aspects of model justification. We distinguish between the justification that the modeller performs for herself and the justification that she possibly performs for the stakeholders. Sometimes, the model has to be explained to the stakeholders in order to convince them that the model is adequate.

Testing the model is a way to validate the model and it can be used as means to both convince the modeller herself and the stakeholders.

Sometimes, the model is not adequate but it is considered good enough for an initial analysis. The compromise is made when the model is representative enough.

A justification argument can also have formal elements. For example, if we are using model-checking we can use the technique to validate the model

itself, by inventing different testing queries that will show whether the modeller designed the model she intended to.

5.5 Summary of the chapter

In this chapter we show the following.

- Taxonomy of modelling steps, which is our main contribution. This classification names explicitly kinds of steps made during model construction. These steps are taken over from analysis made by philosophers of science and for one part from the system engineering methods. Distinguishing different kinds of steps supports reasoning while designing (constructing) the model. At the same time these steps can be documented for later model validation and justification. As such it is part of the method for model construction (Design Problem 1.1) as well as part of the method supporting model justification (Design Problem 2.1).
- Modelling process steps - the steps describing the modelling process as guidance and checklist when designing the model (as solution for the Design Problem 1.1) as well as guidance for model justification (as solution for Design Problem 2.1).

Chapter 6

The initial version of the method: Lego sorter

In this chapter we address design and conceptual problems shown in Table 6.1 (which shows a subset of questions in Table 1.1).

Design problems, conceptual problems and knowledge questions
DP 1.1 Design problem Design a method that guides the **construction** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.
DP 2.1 Design problem Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.

Table 6.1: The top level design problem is decomposed into design problems, conceptual problems and knowledge questions.

Specific about this case is the following.

- This is our first case to validate and identify non-formal modelling steps for questions D1.1 and D2.1.

- It is a small mechatronic system, made in our lab, with components interacting with each other, and with workpieces moving through the system.
- It is complex enough for the first case and at the same time simple enough not to get us into details of other aspects that are out of the thesis focus.
- We model the Controller and the Domain containing the Plant and the relevant aspects of the system environment (the bricks and the user behaviour).

We apply and further refine the steps described in the initial method version in Chapter 5 on this case. Modelling a concrete system enables validation of the first version of the method.

On Figure 6.1 we outline the modelled system.

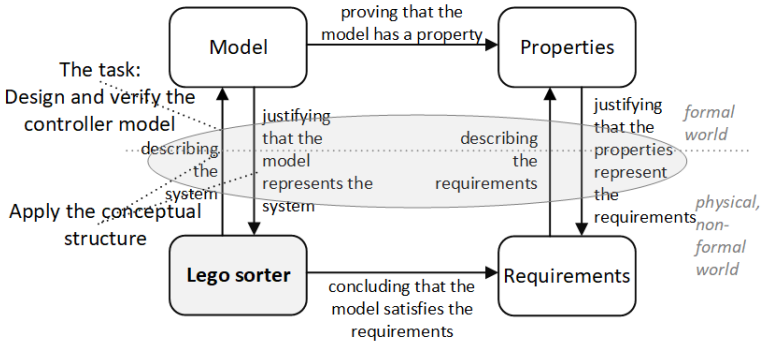


Figure 6.1: Case study: Lego sorter, a simple mechatronic system constructed in our lab. Simple enough to test the first method iteration, and complex enough to be a realistic case.

The result of this chapter is as follows.

- Validated taxonomy of modelling steps. These steps have twofold function - they distinguish different kind of model design steps to achieve model purpose (DP 1.1). At the same time they can be named when explaining the model and validating that it represents the system correctly (DP 2.1).
- Validated and refined modelling process steps - the steps describing the modelling process as guidance and checklist when designing the model (DP 1.1) as well as guidance for model justification (DP 2.1).
- An extension of the justification argument with the concept of modelling assumptions. (D.P. 2.1)
- Identified importance of informal models as a "bridge" between formal model and the system.

We describe the case and then reflect on the results.

6.1 Case Study: Lego sorter

6.1.1 The plant description

The Lego sorter is a small PLC (Programmable Logic Controller)-controlled plant made of Lego bricks, DC motors, angle sensors and a colour scanner. It should sort yellow and blue bricks according to their colour. Figure 6.2 shows a top view of the plant.

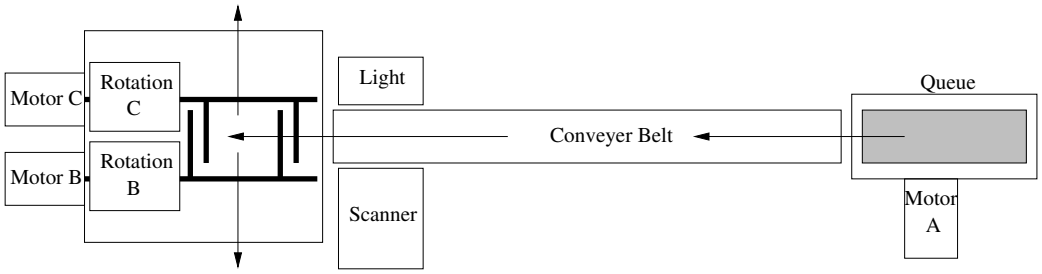


Figure 6.2: Top view of the Lego plant

Bricks are stored in a pile (see fig 6.3), all at once or put individually. Two wheels at the bottom of the queue are moving the bottom brick to the conveyor belt. The conveyor belt transports bricks to the scanner and the sorter. At the scanner, the colour of the brick is observed. The sorter consists of two fork-like arms. Each arm can rotate a brick to one of the sides of the plant.

Bricks are entering the belt one after another, and it is possible to have more than one brick on the belt. The queue is built in such a way that there is always a minimal distance between the bricks on the belt. This is important because scanner would observe bricks that are attached to each other as one single brick.

The wheels of the queue and belt are coupled - this means that one motor is moving them. The scanner is a sensor that senses no brick in front of it, or yellow or blue brick in front of it. Putting a brick of another colour in front of it would cause the system to enter an unknown state. Each sorter arm is controlled by its own motor and has its own rotation sensor that senses the angle of the arm. The starting angle is 0, and as the arm rotates, it changes to 360 degrees.

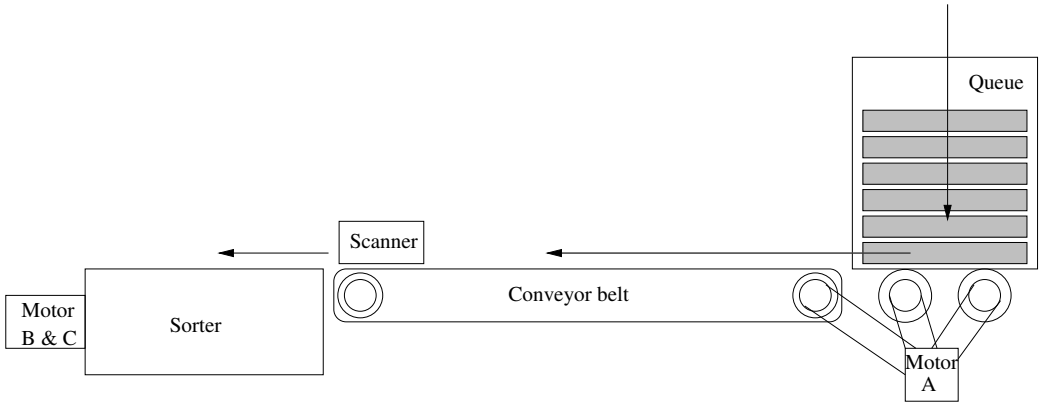


Figure 6.3: Side view on the Lego plant

There are two tasks here: to design a controller and to verify the system correctness. After formulating the requirements in natural language, we will use the problem framing technique to classify the problem, and we will describe it with problem diagrams. We will continue describing the system in more detail with state charts. Then, we will proceed with modelling based on timed-automata, suitable for model checking with the Uppaal (Uppaal 2023) tool, an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata. We will write down the assumptions made while modelling. These activities are not necessarily sequential; it often happens that one stage reveals errors, omitted information or ambiguities in previous stages.

Before we proceed with describing the model and the process, note that this was our first case in which we were the ones giving the modelling task, being the domain experts and the users of the model. So the problem analysis phase only required a few iterations of refining the requirements to verify. For the justification of the model, we explained them to our colleagues, but they were not the users of this model.

6.1.2 Requirements and decomposition

Problem analysis: The purpose of the model

The purpose of the model is to design and verify the controller. The model verification formalism and tool are Uppaal. More on the Requirement and the system fragment being modelled is described below.

Problem analysis: Fragment

In this case, we will describe the whole system at a higher level of abstraction because the system is simple enough to do so. We recognise the problem as one of the elementary problems that fits into the *required behaviour*

Figure 6.4: Problem diagram of the Lego plant and the Machine (Control)

frame (Jackson 2000). The plant is the controlled domain, and the problem is to build a machine that will impose the control. We are not changing anything in the Plant architecture.

Figure 6.4 documents the first decomposition step, where $\mathcal{R}_0$ is a requirement, and **a** is the description of the interface between the Plant and the Machine (control software). The $\mathcal{R}_0$ represents the desired property and, informally speaking, is stated as: "All bricks in the Queue will be sorted eventually by the plant.". The interface description **a**, describes the phenomena shared by the Plant and the Machine.

The verification requirement in its first version is: Plant $\wedge$ Machine $\implies$ $\mathcal{R}_0$

Model design: Decomposition

The first decomposition step is to describe separately the controlled Plant and the Machine that implements the controller. These are the two domains with sensors and actuators at their shared interfaces.

Looking from a perspective of the verification requirement, at this level of abstraction, we need a description with sufficient elements for a formal proof. Looking from a control (control software) designer perspective, we want to decompose further to find out more information about environment and processes in the Plant.

The first refinement step of the system is described in figure 6.5

We decomposed the environment into physical entities called *domains*. We call this decomposition *instrumental decomposition*. In system engineering, this is also called a physical system decomposition. There are other possible decompositions, *e.g. process-based*.

We refined the requirement $\mathcal{R}_0$ to $\mathcal{R}_1$ = "All bricks in the Queue are eventually moved by the Sorter to the side corresponding to their colour." The $\mathcal{R}_1$ is described as **r1** $\implies$ **r2**, where:

r1: Eventually all bricks of the Queue are gone and,

r2: Eventually all bricks are on the correct sides of the Sorter.

Formal vs. informal

We will continue with descriptions of each domain and interface. Our description will be informal but precise.

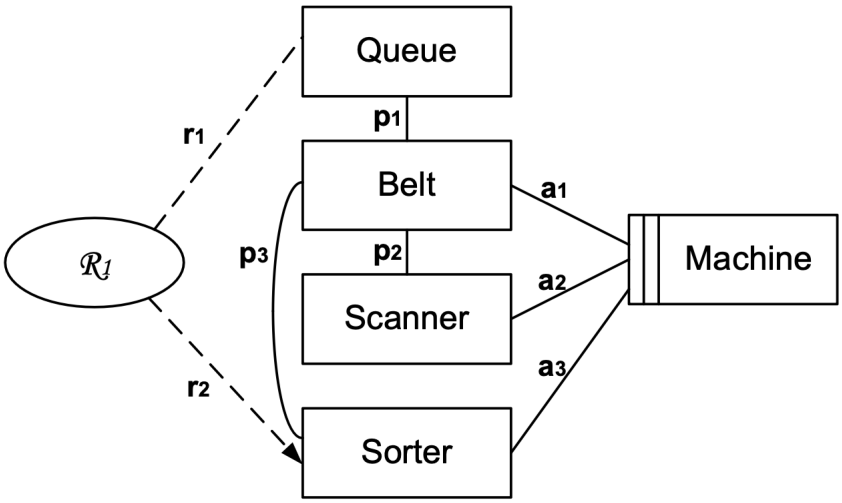


Figure 6.5: Refined problem diagram of the Lego plant and the Machine (Control)

Model design: the Domains

The informal but precise domains descriptions are as follows.

Queue: The Queue is a container that can hold bricks (of any colour). If there are bricks in the Queue, one brick rests on the wheels coupled with the belt.

We assume that bricks will be standard Lego bricks, appropriately stored - they will fill in the stack laying with its longest side, ordered one on the top of each other. There will be a maximum of N bricks.

Belt: The Belt can move, transporting a brick from the Queue to the Scanner. The length of the belt and bricks is such that there can be a maximum of two bricks on the belt. We approximate the belt speed in the following way: The speed of the belt is either v or 0. So, we abstract from the acceleration and deceleration of the belt. A brick is present in front of the Scanner for a minimum time of $t = \frac{l}{v}$ seconds, where v is the speed of the belt (in meters per second), and l is the length of a brick (in meters). We choose to do this because we do not include v in calculating either time or any other variable.

The plant should be started with no brick on the belt or in the sorter.

Scanner: The Scanner can recognise the colour of a brick that passes in front of it. It distinguishes blue and yellow bricks, and the absence of bricks.

If a brick of a different colour is present in front of the Scanner, it will recognise it as either a yellow or a blue brick. There is a minimal distance between the bricks so the Scanner recognises them as separate.

Sorter: The sorter can rotate its arms, but the control should ensure that

they do not rotate at the same time. Which arm rotates depends on which motor is switched on by the controller.

The order of a brick arriving at the Sorter and a new block arriving at the Scanner is nondeterministic. If the first block passes the Scanner and the next arrives, the control will stop the belt. The sorter should start with the proper initial position so that its arms do not block the brick arriving from the belt.

Machine: The Machine can interact with the Belt, Scanner and Sorter through interfaces **a1**, **a2** and **a3** (described below). Through these interfaces, it should behave as follows.

The Machine continuously receives one of the three possible values from the Scanner.

The Machine switches the Belt on, if there is no brick at the Scanner.

The Machine switches the Belt on if the Sorter is idle. It switches the Belt off, if there is a brick in the Sorter and a brick at the Scanner.

The Machine causes the Sorter arms to move according to the colour of the brick in the Sorter. After it receives a full rotation angle from the rotation sensor, it switches off the sorter arm.

Note that for this simple case we did not need to control dynamical Plant aspects. The controller is a discrete one, switching on and off the motors.

Interfaces The interfaces description, informally, are as follows.

a1: The Machine switches the Belt on and off.

a2: The Scanner sends one out of the 3 values to the Machine - blue, yellow or nothing.

a3: The Sorter sends the angle positions of the arms to the Machine; the Machine switches each sorter arm on and off.

p1: The first brick (yellow or blue) in the Queue moves to the Belt, or nothing moves to the belt.

p2: The Scanner detects whether there is either nothing or a yellow brick or a blue one at the scanner position.

p3: If a brick is at the end of the Belt and the Belt is moving (and the sorter is idle), then the brick moves to the Sorter.

The verification requirement at this stage has the form
$\mathsf{Queue} \wedge \mathsf{Belt} \wedge \mathsf{Scanner} \wedge \mathsf{Sorter} \wedge \mathsf{Machine} \wedge \mathsf{Natural\_Laws} \implies \mathcal{R}_1$,
where all elements should be filled in by a description that should formalise the informal description above. Note that in the verification requirement we do not have explicit interfaces. The composition is done by the logical $\wedge$, and all phenomena taking place on the interfaces should be part of the domain descriptions sharing the interface. (This also coincides to the way (Gunter et al. 2000) describes the verification requirement.)

The Natural_Laws typically contain facts as: If the belt is not moving a brick on the belt remains in its position. or *A brick does not change its colour.*

Following the notation described by (Jackson 1995, Jackson 2000) we write down the previous description as follows.

a1: MACHINE! {belt_start, belt_stop}

a2: SCANNER!
{nothing_at_scanner, yellow_brick_at_scanner, blue_brick_at_scanner}

a3: MACHINE!
{yellow_arm_start, yellow_arm_stop, blue_arm_start, blue_motor_stop}
SORTER!
{yellow_sorter_angle, blue_sorter_angle} (the latter are reals (or integers), the rest are booleans)

p1: BELT!
{blue_brick_moves_to_belt, yellow_brick_moves_to_belt, nothing_moves_to_belt}

p2: BELT!
{blue_brick_is_at_scanner, yellow_brick_is_at_scanner, nothing_is_at_scanner}

p3: BELT! {blue_brick_moves_to_sorter, yellow_brick_moves_to_sorter}

Dictionary

The description of the interfaces becomes formal only after we create a dictionary of the terms involved. The interfaces may contain states, events or values. No matter how intuitive the names of the interface phenomenas are, we cannot distinguish which one is a value and which one is a state. This is important because our next step will be a formal, state machine description. We will map the phenomena on the interfaces to the events and states in the state machines. Therefore, it is important to know, for example, if the 'belt_start' is an event of turning the belt motor on or a state that indicates that the belt is moving.

In this problem diagram, all the interface phenomena are events except values of sorter arms angles ('yellow_sorter_angle', 'blue_sorter_angle').

6.1.3 Model design: State charts - based domain descriptions

Problem diagrams represent a decomposition to domains and the phenomena on their interfaces. They are the static picture of the system.

For the description of the domains' behaviour, we choose the state charts description (Wieringa 2003), (Harel 1987). This was to explain the model to colleagues who acted as model stakeholders and did not understand Uppaal. The downside of this is that there is a possibility for an error, due to the fact that the semantics of Uppaal and Statecharts are different and there was no tool to automatically perform the transformation.

First, we list all events to which the environment responds and that are most likely to happen. This way, we will, as it was already explained in Chapter 5, describe all possible relevant behaviours. They are the events on the domain interfaces with the Machine and the other domains. When listing these events,

we assume that some events will certainly not happen, like, for example, turning the belt on in the wrong direction. This is a finite list.

Second, for each event, we define conditions under which they can occur, so we consider only a few state variables. Third, for each event[condition] pair, we define how the plant responds (assuming that each device works normally).

State charts represent all possible state transitions in every 'possible order'.

Queue For each domain, the events that we take into account are events on its interface, namely on the interface with the Machine and on interfaces with other domains. The Queue domain (see Figure 6.6) is the domain without an interface with the Machine. (Another solution within this decomposition was to put the Queue as a part of the Belt domain.)

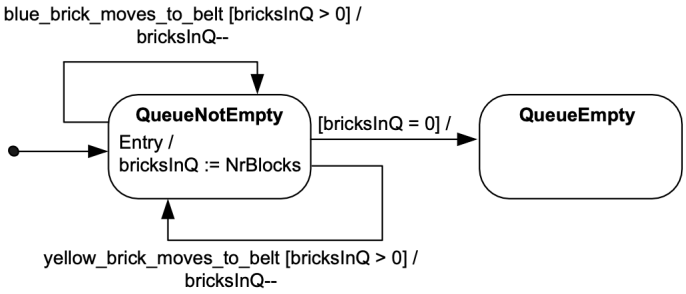


Figure 6.6: State diagram of the Queue

Belt The figures 6.7, 6.8, 6.9 and 6.10 show the Belt description. This version has independent descriptions of Belt interfaces with the Scanner and the Sorter. We have also designed another version with the two interfaces described together. After comparing these two models, we chose the first solution because of its better clarity of representation. The price of this decision is greater state space.

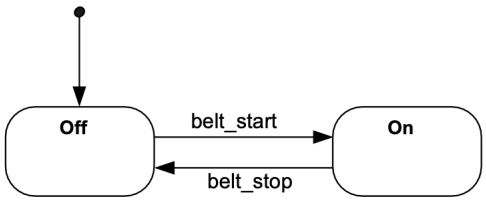


Figure 6.7: State diagram of the Belt

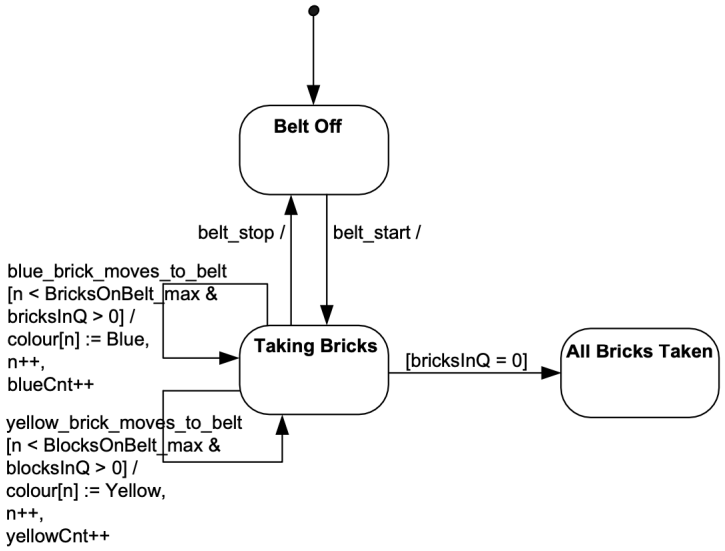


Figure 6.8: State diagram of the Belt

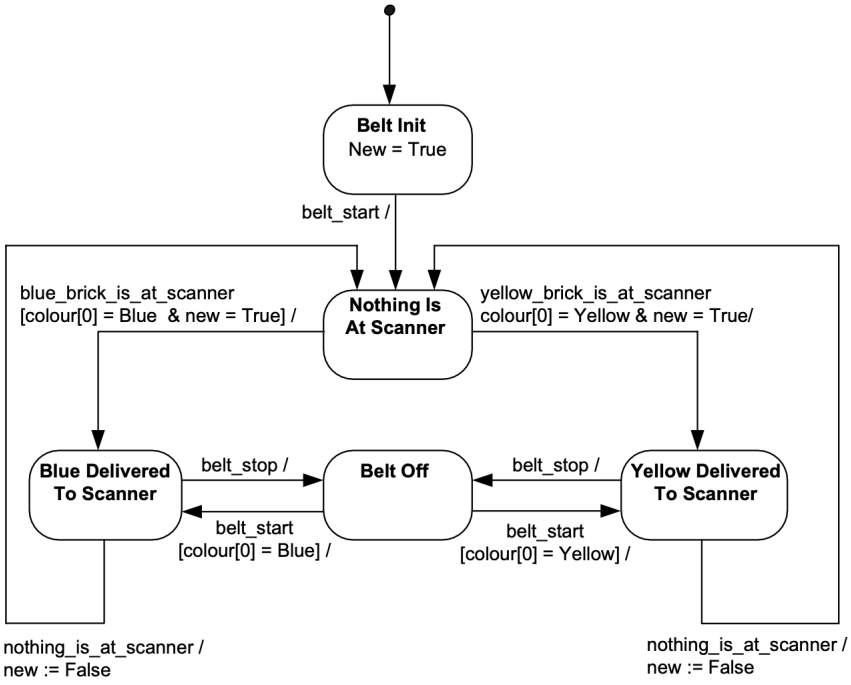


Figure 6.9: State diagram of the Belt

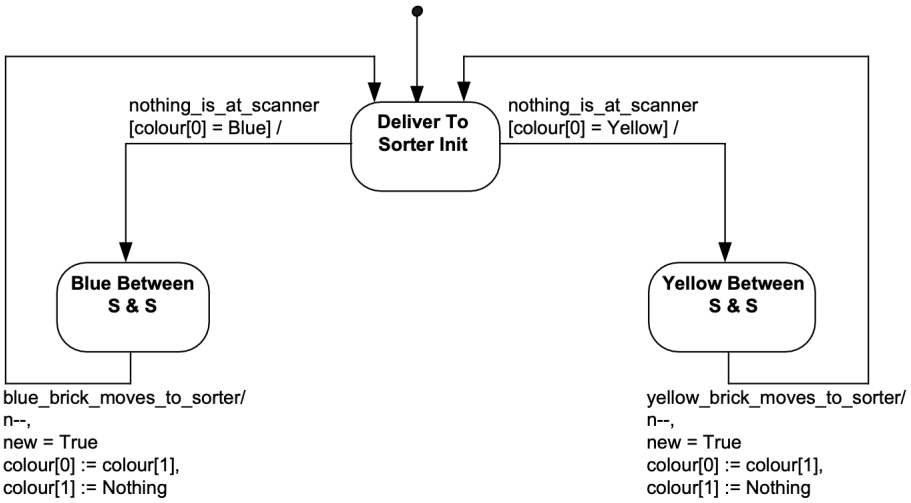


Figure 6.10: State diagram of the Belt

Another reason to describe the interfaces separately is the possibility of changes in the model. In this model, a brick goes from the Queue to the Scanner, from the Scanner to the Sorter. If we want to add more components with interfaces to the Belt, we might do that with adding a new state machine instead of changing the previous one (have to think more about this, it is quite a strong statement).

Scanner The Scanner state diagram describes phenomena on the interfaces with Belt and the Machine (figures 6.11).

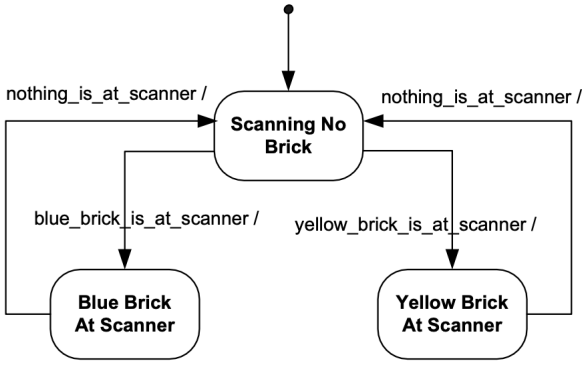


Figure 6.11: Scanner state diagram

Sorter The Sorter state diagram describes phenomena on the interfaces with Belt and the Machine (figures 6.12).

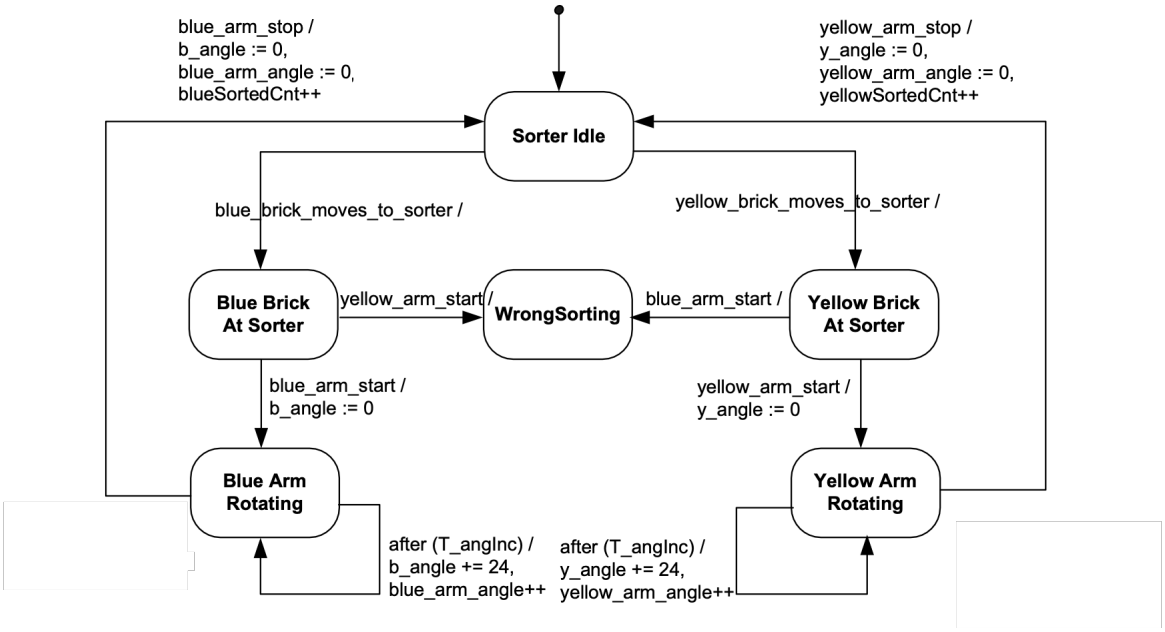


Figure 6.12: Sorter state diagram

6.1.4 Design and specification of the control

Control design

The problem description does not require control engineering techniques to be applied. We can analyse the plant and define a control without them.

We are designing a control for the plant model decomposed to a collection of Queue, Belt, Sorter and Scanner and their interfaces. At this level of decomposition and due to the simplicity of the case, we design a control specification based on *insight*. Whether the control designed as result of insight and the plant satisfies the requirement will be verified during the model checking phase.

As we can see on the problem diagram shown on Figure 6.5, the Belt has three interfaces with other parts of the plant - it is taking the brick out of the Queue, transporting it in front of the Scanner and putting it into the Sorter.

The two sub-controls should be designed - controlling the Sorter and the Belt. The first one is straightforward - the Sorter starts sorting when it is calculated that a brick is in the Sorter, it stops after a sorter arm makes a full rotation, sorting brick at the proper side of the plant.

The other sub-control is controlling the Belt. Care has to be taken that the next brick does not arrive to the Sorter while the current one is being sorted.

One possible solution is to stop the Belt when a brick arrives at the Sorter and to start it again when the Sorter finishes sorting. Another solution is to keep the Belt running all the time, except when a new brick is observed at the Scanner and the current one is not already sorted.

The first solution is more time efficient, because while the sorter is sorting, the belt can bring a new brick to the Scanner. We chose the first solution so the control described in informal language is:

sc1: If the Sorter is idle, the Belt is moving.

sc2: If nothing is at the Scanner, the Belt is moving.

sc3: If the Sorter is not idle and there is a brick in front of the Scanner, the Belt is stopped.

sc4: If a brick is in the Sorter, the Sorter is sorting, until one of its arms makes a full rotation.

The control described with these sentences is something that, hypothetically speaking, can be implemented in various ways. We do not know whether we have a person switching on and off the motors and reading sensors, or maybe mechanical devices doing this task.

Control software description

As software engineers, we have to implement control that is designed. When designing verification model we stay in high level system descriptions (Mader 2000), which means that we do not specify a PLC program at this stage; we will describe control software with state machines and later, with timed automata .

For the design of the control software, we **were influenced** by the environment decomposition. Looking at the Belt, Scanner and Sorter we decomposed the control to subcontrols controlling these parts.

All the motors are DC motors and we control them by turning them on or off. We can describe each subcontrol as function of discrete events and states of other domains. For both the Belt and Sorter control we need to observe brick arrival at Scanner and value of rotation angle sensors.

Figure 6.13 is a problem diagram - like sketch of the control, showing what is controlled and what information is exchanged between different parts of the control.

On the interfaces there are either values of shared variables (colour_to_sort) or actions on which processes are synchronised (all the other elements of interfaces):

c1: SCANNER_CTRL!
{nothing_at_scanner, yellow_brick_at_scanner, blue_brick_at_scanner}

c2: BELT_CTRL! {start_timer}

c3: DELAY! {start_sorter}

c4: BELT_CTRL! {colour_to_sort}
SORTER_CTRL! {sorted}

The sorter signal depends on a colour detected at the scanner (this determines which motor will be turned on), the scanner signal observing a brick, and the angle read on the rotation angle sensor. We do not have a brick detector in

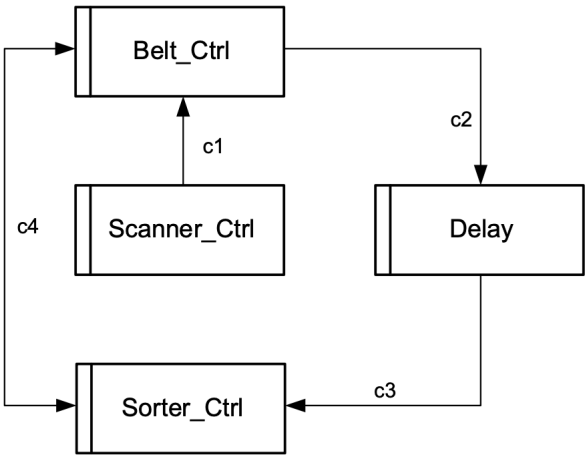


Figure 6.13: Control decomposed

the sorter, but we measured the maximal time for a brick to arrive to the sorter after it was detected at the scanner.

State diagrams on figures 6.14, 6.15, 6.16 and 6.17 show the solution for the control, i.e. software.

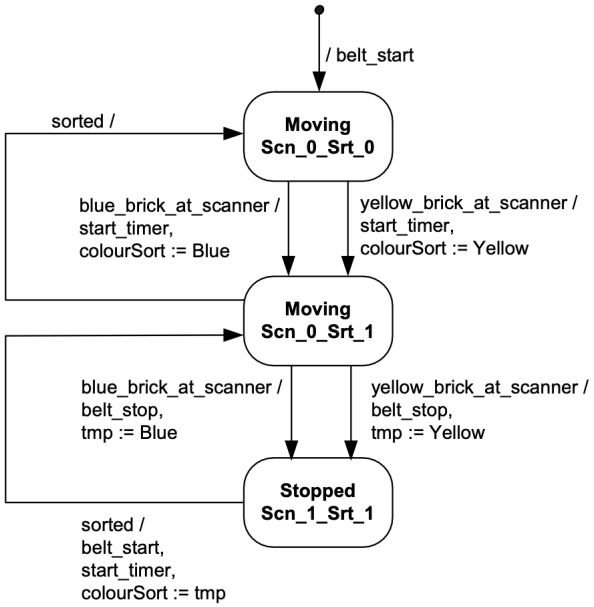


Figure 6.14: Belt control process

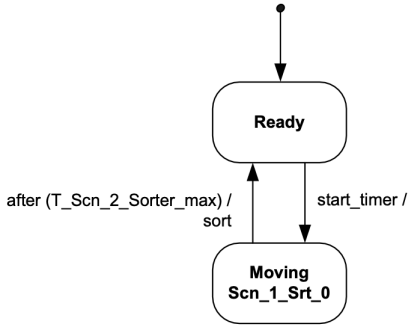


Figure 6.15: Delay process

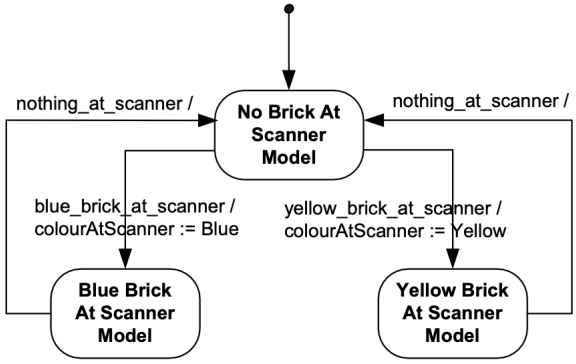


Figure 6.16: Scanner control process

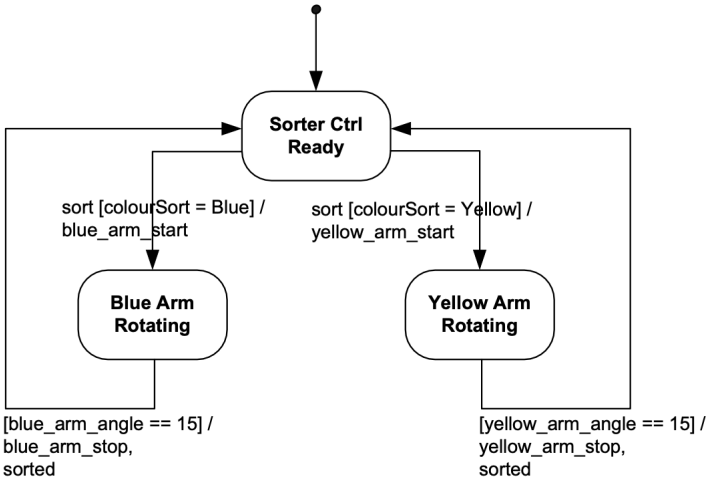


Figure 6.17: Sorter control process

6.1.5 Modelling in Uppaal

Uppaal is a toolbox for validation and verification of real-time systems (Behrmann et al. 2004). A system is modelled with network of processes represented with a timed-automaton each. Timed automata are finite state machines with time (clocks) composed of locations and transitions between them. We can perform a simulation by running the system interactively to check if it works as intended (this is just a sanity check). The verifier checks reachability properties, i.e. if a particular state is reachable or not.

When we derive the Uppaal model from the state charts, there are rules we applied straightforwardly and there are modelling decisions related to the tool itself. The rules that we apply automatically are:

- States from state charts are translated to timed-automata locations.

- Events from state charts become channels in Uppaal.

Internal descriptions are shown through variables that are updated and conditions for transitions (guards).

The additional modelling decisions are and Uppaal language specific characteristics are as follows.

- In our model, control is not an infinite loop as is the case with the most applications made for reactive systems, but there is an end state. This end state is seen as deadlock from the verifier perspective, so we add transition in one of the final states of the automata to avoid deadlock report.

- Committed locations are characteristic of Uppaal models and we used them to describe atomic actions.

- If there is a location that is not committed and a channel that is not urgent, the system may stay in this location forever. Therefore an invariant is added so that the system can leave the location.

- If a channel a is a broadcast channel, than $a!$ can be fired even if there is no any $a?$ to synchronise with.

- Uppaal has the explicit notion of time, and if we want to describe that one event will come after another, we have to use clocks and define the exact time between these two events. For example, if we want to describe that a brick is moving from the queue to the sorter, we have to use clocks and define a time that passes between an event of "brick got out of the queue" and "brick arrived at the scanner".

If we have a location in the Uppaal model without a guard and a transition without an invariant, this transition may never happen.

Environment

Figures 6.18 and 6.19 show the Uppaal implementation of the Queue state chart. The auxiliary automaton is one of the 'tricks' we need when modelling in Uppaal to enable transition (channel) x.

Figures 6.20, 6.21, 6.22 and 6.23 represent the timed automata of the Belt and its all possible responses to events on its interfaces. The event noth-

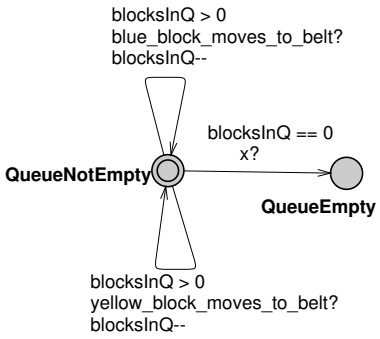


Figure 6.18: Uppaal model of the Queue

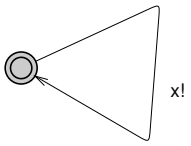


Figure 6.19: Auxiliary automaton

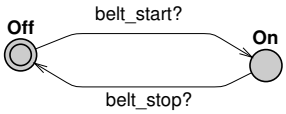


Figure 6.20: Uppaal model of the Belt moving or being stopped

ing_goes_to_belt exists only because of the Uppaal characteristic mentioned above (we do not want a deadlock when an end state is reached).

The Uppaal model of the Scanner (figure 6.24) has a committed location because we assumed that event of scanner signalling a brick arrival to the Machine happens in infinitely short time after actual brick arriving at the Scanner.

On the figure 6.25 the Uppaal model of the Sorter is shown.

Control

The model of the process controlling the belt (figure 6.26) has the same structure as the state chart, but there are again committed locations that describe actions (channels) that occur at the same time (that are atomic).

The process that triggers the sorter to start sorting is the Delay process, shown on the figure 6.27.

It has variables that keep the information about the colour observed; these are so called model variables - they represent the Scanner in the control.

Finally, figures 6.28 and 6.29 show the Uppaal model of the scanner and sorter control processes.

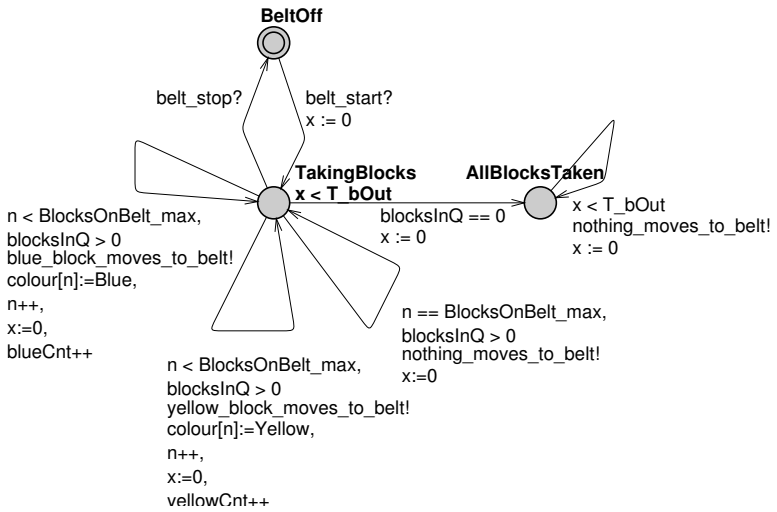


Figure 6.21: Uppaal model of the belt taking bricks from the queue

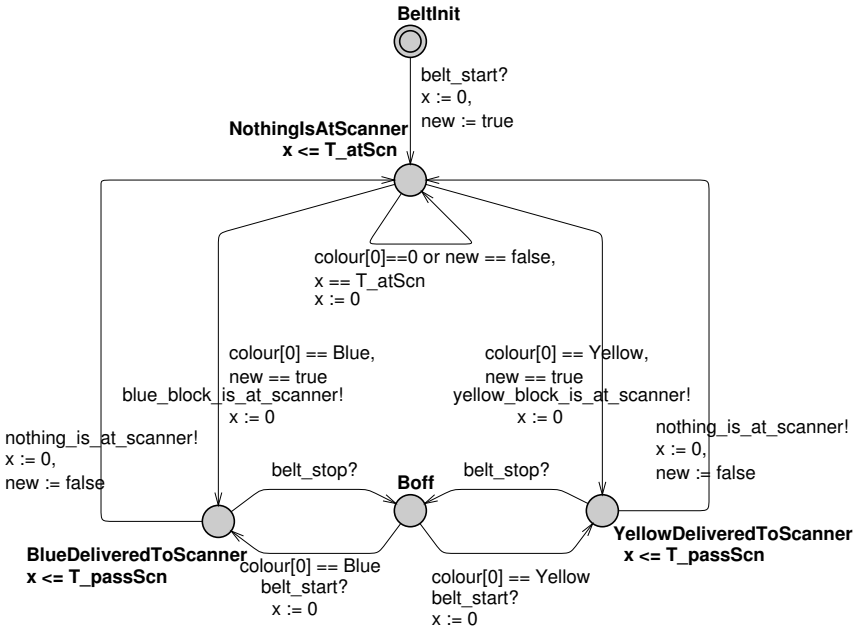


Figure 6.22: Uppaal model of the Belt transporting brick to the scanner

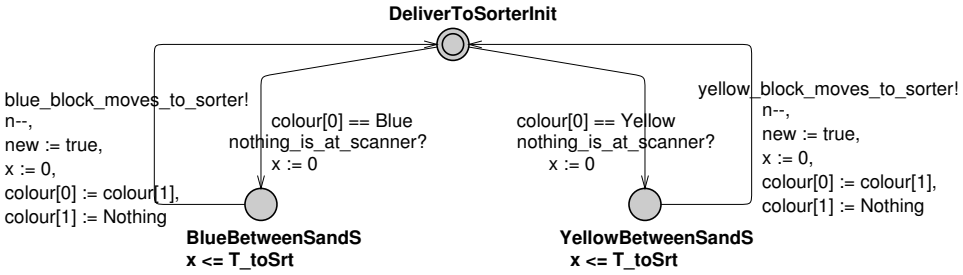


Figure 6.23: Uppaal model of the Belt transporting brick to the sorter

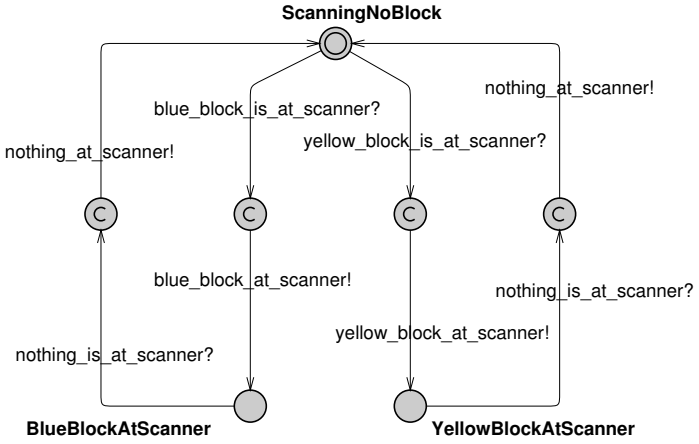


Figure 6.24: Uppaal model of the Scanner

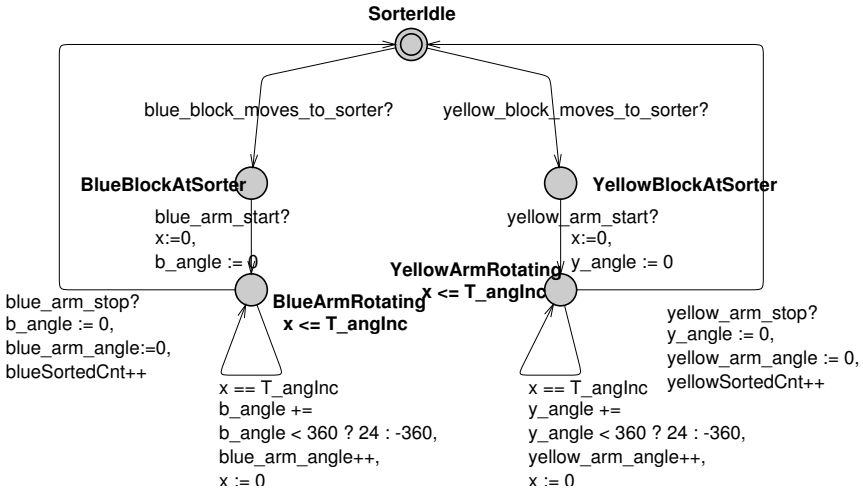


Figure 6.25: Uppaal model of the Sorter

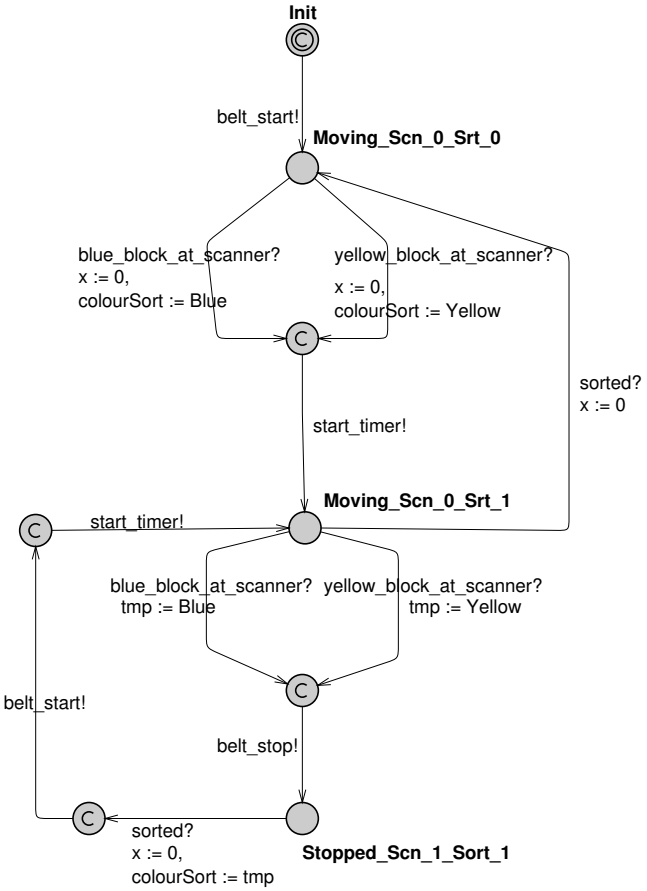


Figure 6.26: Uppaal model of the Belt control process

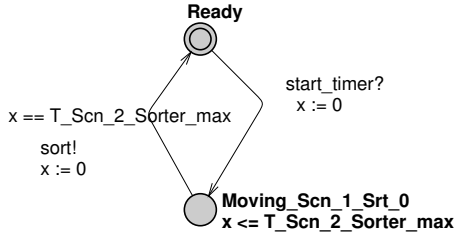


Figure 6.27: Uppaal model of the Delay process

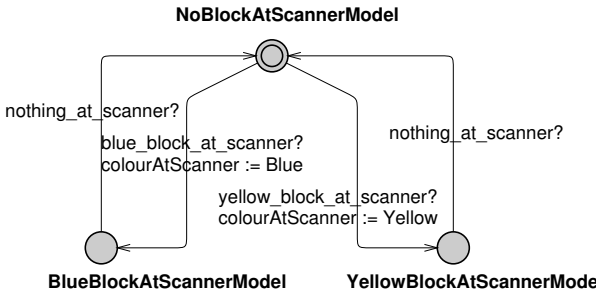


Figure 6.28: Uppaal model of Scanner control process

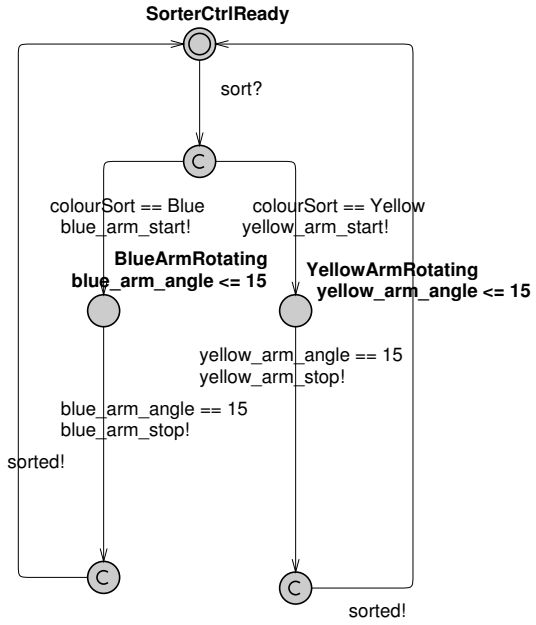


Figure 6.29: Uppaal model of Sorter control process

6.1.6 Model justification: Documenting assumptions

A model is an abstraction of reality and keeps only the information that we find necessary and relevant to the property we want to verify. Still, there is some relevant information that is not in the model, additional conditions. These are the *assumptions* about the environment. It is very important to document them, and experience shows that the assumptions that seemed 'obvious' to the modeller or designer were not obvious to other people. This can cause faults in the system.

There are two kinds of assumptions we made. The first is made in order to prove that the requirement is satisfied. These are the assumptions that need to be tested when we move control to another environment, i.e. when we change something in the environment and plant. There are also assumptions made because we want to use a particular tool. These assumptions have to be checked when we move to another tool.

We list assumptions that we found important and it still might be the case that we have omitted something. It is difficult to track down all the assumptions made, even after the back-and-forth steps of refining three kinds of descriptions (informal language description, state machine description and Uppaal model). Some of them seemed obvious, like "We start with empty belt and sorter." and it turned out to be the first thing that the users of the sorter tried. The most obvious assumptions are the first to forget and only after a lot of testing we can be reasonably sure that we have listed them all. Still, there has to be a balance between really important assumptions (that only seem obvious to one person and are not to the other) and trivial ones. The assumptions are as follows.

A1: Bricks are standard Lego bricks (50mm x 15mm x 7mm) that fit in the Queue.

A2: They are placed correctly in the Queue, either one by one or in a pile.

A3: The plant is started with a brick laying neither on the Belt nor the Sorter.

A4: There are only blue and yellow bricks in the Queue.
If there is a brick of another colour, it will be recognised as either yellow or blue and sorted to one of the sides of the plant, but the model does not describe this property.

A5: The number of bricks that are put in the Queue is N; we do not have an operator that puts a number of bricks not known to us.
This is one of the assumptions that is not close to practice, where there is a "while(forever)" loop of control that controls sorting of bricks. At the end of the day, when someone pushes shut-down button, the control enters the finalisation mode and shuts down everything in a proper way (or for more robust system, you just switch off the power).

A7: There is a minimal distance between bricks so that

A7.1: There is always "nothing_at_scanner" observed by Scanner before the new brick is observed and

A7.2: There will be no new brick in front of the Scanner before a previous brick moves to the Sorter.

A8: The order of events of the leading edge of a brick observed and a new brick entirely on the belt is not deterministic. It can happen that a brick arrives in front of the scanner before previous brick is sorted

A9: Sorter arms have proper initial position, as it is shown on the figure 6.30. Proper initial position is a common problem in mechatronics applications. In practice, people never assume this. Instead, there is always a trick in initialisation mode to move or rotate something until the control recognises the current position.

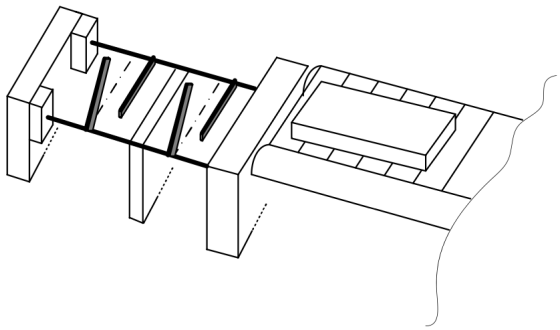


Figure 6.30: Irregular position of a sorter arm that stays on the way of the brick

A10: All parts in the connection domain (motors and sensors) are working properly and introduce no delay

A11: Scanner observes the colour all the time (level-based, not edge-based)

A12: PLC control is fast enough to 'catch' rotation sensor steps (and all other things, but this one is the most critical)

The assumption that "The machine is fast enough to keep track of angle increments." belongs to the group of assumptions called *perfect technology assumption* which means that the processor is fast enough and that hardware components of our embedded system are working sufficiently fast.

Some of these assumptions, if not fulfilled, will have a faulty system as consequence, like for example *A7*. For some of them, we can say that if they are not satisfied, we can not guarantee that the system is working correctly (but it might be the case that it works). For example, if *A9* is not fulfilled, there might

be a sorter arm directly on the way of the brick, which causes fault, or the arm is slightly moved, but still not on the way of the brick and the system can still work correctly.

Another group of questions to ask about each assumption is "Can it be violated? By what/whom?" and "Can violation be prevented? By what/whom?".

We also list the natural laws:

N1: If on the Belt for T1 seconds, and the Belt is moving, the brick changes its position.

N2: If the Belt is stopped, and the brick is on it, it won?t change its position.

N3: The Belt speed v is a function of the value m put on the Belt motor: v=kxm

N4: A brick does not change its colour.

N5: Bricks cannot overtake each other, in the order they move out, they arrive to the scanner.

N6: The plant has to be put on a flat surface, in order not to have gravity force moving bricks

Whether to put them on the assumptions list is a subjective decision.

There are modelling decisions that look like assumptions, such as: The wheels at the bottom of the Queue are seen as part of the Belt.
This is because they are coupled with the Belt by the same motor, so they are stopped and moving at the same time. We might want to look at them separately in case there is a possibility that a wheel does not work.

6.1.7 Model design: Formal verification

The requirement R1 is translated into the property of bricks being sorted in appropriate sides. Since we have automata-based model, we are checking if the system model will reach the state where all bricks are sorted. In the Uppaal we write this in linear temporal logic:

A<> BeltFromQueue.blueCnt == Sorter.blueSortedCnt and
BeltFromQueue.yellowCnt == Sorter.yellowSortedCnt and
BeltFromQueue.blueCnt + BeltFromQueue.yellowCnt == NrBricks

For all paths it will eventually hold the following:
(1) Number of the blue (yellow) bricks that were moved out from the Queue is the same as number of the blue (yellow) bricks on the side of the sorter that sorts blue (yellow) bricks and
(2) The sum of the sorted blue and yellow bricks is the same as the initial

number of the bricks put in the Queue.
The second statement prevents query to be true at the beginning when counters are zero.

We also checked some other properties in order to validate our model. We found it necessary because modelling progression of a brick in time was not easy thing to do in Uppaal. As we mentioned before, our requirement is functional, not a timing property, so we did not put accurate times between events of brick getting out of the queue, passing the scanner and moving into the sorter. However, we wanted to include all possible orders of events and exclude those that are not possible. A brick is not explicitly described in the model, we see its progression in the models of each domain in the plant. So, we tuned times in different automata in order to show the scenarios shown on figures 6.31 and 6.32.

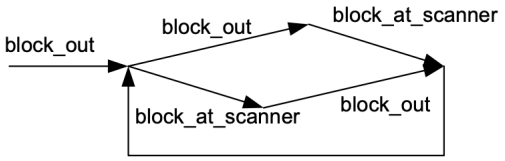


Figure 6.31: Order of brick moving out of the Queue and brick moving to the Scanner

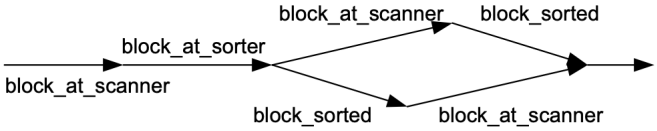


Figure 6.32: Order of brick moving to the Scanner and moving to the Sorter

The queries in the Uppaal were:
E<> Sorter.SorterIdle and (Sorter.yellowSortedCnt > 0 or Sorter.blueSortedCnt > 0) and bricksInQ > 0 and Scanner.ScanningNoBrick and Belt2ScannerAndSorter.NothingIsAtScanner
Scenario 1: A brick is sorted and the new brick has not yet arrived at the scanner.

5. *E<> (Sorter.BlueArmRotating or Sorter.YellowArmRotating) and (Scanner.BlueBrickAtScanner or Scanner.YellowBrickAtScanner)*
Scenario 2: Brick is in the sorter and a new brick arrives at the scanner.

6. *E<> (BeltFromQueue.blueCnt + BeltFromQueue.yellowCnt) - (Sorter.blueSortedCnt + Sorter.yellowSortedCnt) == 1 and (Belt2ScannerAndSorter.BlueDeliveredToScanner or Belt2ScannerAndSorter.YellowDeliveredToScanner)*
Scenario 3: A brick has arrived at scanner, but no new brick has moved out to the Queue.

7. *E<> (BeltFromQueue.blueCnt + BeltFromQueue.yellowCnt) - (Sorter.blueSortedCnt + Sorter.yellowSortedCnt) == 2 and (Belt2ScannerAndSorter.BlueDeliveredToScanner or Belt2ScannerAndSorter.YellowDeliveredToScanner)*
Scenario 4: A brick has arrived at scanner, and new brick has moved out to the Queue.

These are some additional properties, in order to check if actuators can be turned on/off when they shouldn't be.

8. *E<> Scanner.ScanningNoBrick and Sorter.SorterIdle and BeltFromQueue.blueCnt > 0 and BeltAll.Off*
Checking for illegal state: Can the Belt be off if there is a sorter is idle and no brick in the scanner and it is not an initial state of the system?

Update on the justification argument Based on these findings, our justification argument is as follows:
(If the Model (M) satisfies Properties (P)) and (the Assumptions (A) hold (are true)), then (the System (S) satisfies Requirements (R)).

6.2 Summary of the chapter

Using the first iteration of our method we designed and verified the plant model of a simple system - Lego sorter. During the problem analysis phase, we stated the purpose of the model and decided on modelling languages and formalisms to use. To understand how the system works, and later to structure our model in the Model design phase, we used problem diagrams to apply the top level structure of the domain that consists of Plant, Controller and relation of the Plant to the Requirements.

In the model design phase, we created both state machine models and Uppaal models. We created the first ones, to be able to explain it to colleagues who did not know Uppaal.

We validated the steps proposed in Chapter 5, and in the justification process we found out that the modelling assumptions play a role in the model justification. We reiterated, refined the steps designed in Chapter 5 and add the capturing assumptions as an additional step to our method.

In this example we used both Uppaal models and state machines. The state machines were useful to explain the models to colleagues who do not know Uppaal. However - creating them required quite some effort, and inserted another risk in justification, because we created them manually from Uppaal models. This way, the model has been transformed from one formalism to another. The system decompositions made using the problem frames technique were also useful for explaining the model.

After this case, our method consists of the following ingredients

- Taxonomy of modelling steps that provide classification of modelling steps. We add a step of creation of informal models for the purpose of model explanation and justification. This step introduces at the same time a risk, because we have now two new informal relations - apart from the relation system-formal model, now we have to justify correct relation "represents" between the formal and informal model and between the system and informal model.
- Modelling process steps - the steps describing the modelling process as guidance and checklist when designing the model (DP 1.1) as well as guidance for model justification (DP 2.1).
- An addition of the notion of assumptions to the justification argument.

On Figure 6.33 we outline the ingredients of the method after this case.

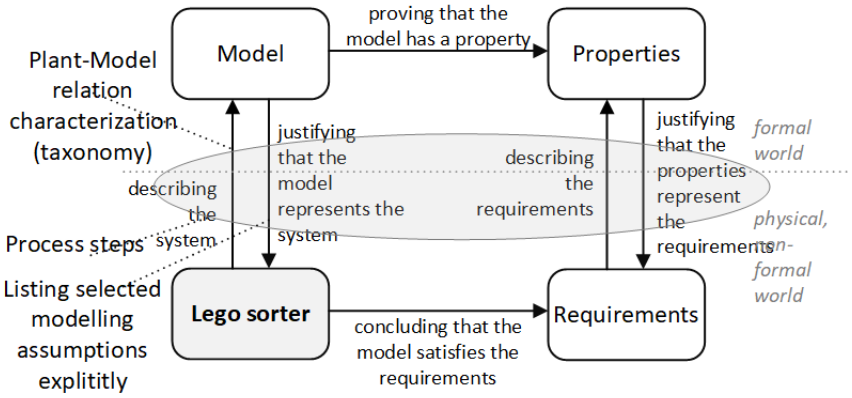


Figure 6.33: Case study: Lego sorter results, including the documentation of modelling assumptions.

Chapter 7

Reusing modelling decisions: modelling a paper inserter

In this chapter we address design and conceptual problems shown in Table 7.1 (which shows a subset of questions in Table 1.1).

Design problems, conceptual problems and knowledge questions
DP 1.1 Design problem Design a method that guides the **construction** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.
DP 2.1 Design problem Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.

Table 7.1: The top level design problem is decomposed into design problems, conceptual problems and knowledge questions.

Specific about this case study is the following.

- This is an industrial case study, performed for a commercial system, and in a context of the organisation that creates it.
- The system is more complex than our first case (Lego sorter).
- The industrial counterpart wants a model that is reusable for the future system generation.
- The domain experts are engineers in the organisation.
- The main user of the model knows the modelling formalism and will be using and maintaining the model.
- The model is the simulation model, not a formal verification model. The model models the Plant.
- The model is created using state machine formalism and the domain experts understand this formalism.

We apply and further refine the steps described in the updated method version described in Chapter 6.

in Figure 7.1 we outline the modelled system and what the modelling task is.

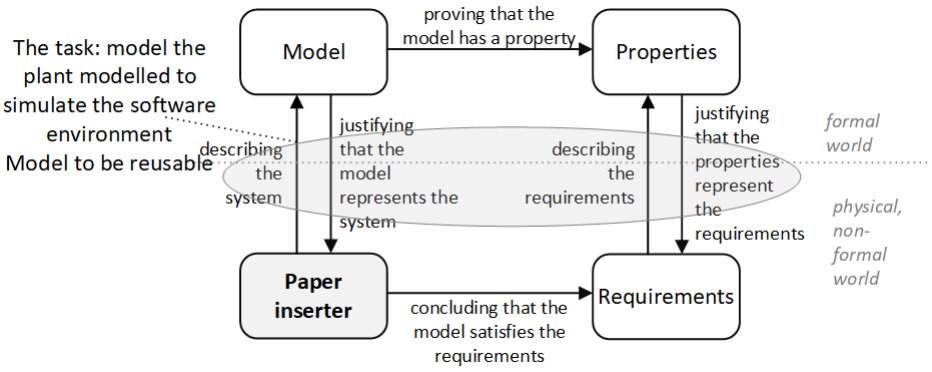


Figure 7.1: Case study: Paper inserter

In the following case, the task was to design models that are reusable. We validated the method, and we used the steps to explain the modelling decisions to the stakeholders. This covers design problems DP 1.1 and DP 2.1. We identified the modelling assumptions, and confirmed that they are an important part of the justification argument.

The result of this chapter is as follows.

- Validated (confirmed) taxonomy of modelling steps provide classification of modelling steps validated.

- Validated and refined modelling process steps - the steps describing the modelling process as guidance and checklist when designing the model (DP 1.1) as well as guidance for model justification (DP 2.1).
- Validating the step of documenting the modelling assumptions. (DP 2.1)

7.1 Case study

7.1.1 The goal of the case study and its relevance for our research questions

The goal of the case study was to validate our method. Given the problem of the industrial counterpart, which was to capture reusable modelling solutions, another goal was to find out what non-formal modelling decisions can be re-used.We started with our high-level classification of modelling decisions and used it as a framework for analysis of the model. During the analysis we adapted it to a framework for capturing re-usable modelling decisions.

Our task was to analyse a plant model made for a system that was under development at a time. More precisely, our task was to, based on the existing model, design a handbook that would assist modelling the plant for the next system generation. To analyse the data we interviewed the designers of the plant model and the model's stakeholders: the project manager, the head of the software department, mechanical engineers, the system integrator, and test engineers. To elicit the modellers' knowledge, we asked for the rationale of their modelling decisions as well as the history of the model development. We also had a hands-on experience with the model.

We performed a case study in a company (*Neopost Technologies*) that develops, produces and distributes different types of mailroom equipment and document systems. The users of document systems are companies that send a lot of paper mail on a daily basis, such as insurance companies, post offices, and banks. One such machine, the inserter, is shown in Figure 7.2. The inserter automatically folds paper sheets and inserts them into envelopes.

The inserter is a typical example of a system which plant consists of mechanical elements whose connection and synchronisation are relevant for the software verification. Our research is aimed at these types of the machines, and the case studies we performed include a lab-made sorter of blocks and a printer. These are all the devices manipulating products (papers, blocks), highly modularised, controlled by real-time embedded software.

Apart from our lab-designed case, two other cases (this one and that of a printer) are performed in a large companies that have subsidiaries in different countries, and that have large software development departments. They also produce their devices in cycles, and do not preserve knowledge from the current system generation for the use in modelling the next system generation.

The company in case has the organisational structure as follows. There is the division to the departments responsible for mechanical, electrical and software parts, respectively as well as the system engineering department. Orthogonal to

Figure 7.2: The inserter folds paper sheets and inserts them in envelopes.

this division, teams are assigned to different projects. The head of the project is responsible for managing different product development phases, whereas the head of a department is responsible for providing relevant expertise to different projects.

One of the emergent benefits of the simulation model that we will describe is that it brought together different domain experts in early phases of the development. Their communication is crucial in these early stages, and shortens the time to integrate components once they are designed and produced.

7.1.2 System description

The inserter works as follows. The operator places documents (paper sheets) and envelopes in appropriate feeders. Through the user interface, he specifies options (recipes) for manipulating documents. A recipe defines how many documents will be inserted in each envelope, whether the paper sheets will be folded, and if so, how; a document can be folded in half or into thirds (in Z or C shape). An example of a recipe is: "Select three documents from the first feeder, fold them in half and insert them into an envelope." Recipes are combined into scenarios, which automates further the inserter's work.

The diagram in Figure 7.3 shows the inserter's modules. Each module has its own function, which is to perform a certain process on a paper or an envelope.

The arrows in the diagram represent the flow of the documents and envelopes through the system. In the feeder, rollers (small plastic wheels) pull out the first document on the pile. The rollers and a transport belt move the document forward and bring it to the collator. The collator collects the documents that belong to the same envelope and collates them. After that, the documents are moved to the folding position. A stroke of a long, thin, sharp-edged arm folds the documents. In the meantime, the rollers of the envelope feeder pull out the top envelope. The rollers along the envelope transportation path bring the envelope to the flap moistener. The flap is first turned up and then moistened with a stroke of a brush. When the envelope takes the position in the inserting module, its upper side is slightly lifted, to enhance document insertion. Folded documents are then inserted in the envelope, the flap is closed and the rollers move the envelope with documents toward the exit.

Every process performed on documents and envelopes is the result of controlled movements of mechatronic parts. Rollers, folding arm, brush and many other system parts are connected to actuators that move them. Along the system, sensors are placed to signal the presence or absence of material in front of them. Based on sensor readings, the control software sends signals to actuators.

The high-level system architecture is shown on the left side of Figure 7.4. The System Controller executes operator requests and recipes, forwarding them to the Embedded Controller. The Embedded Controller controls the plant.

7.1.3 Plant and control integration

When a new generation of the inserter is designed, some mechanical parts, or even whole modules, are re-used from the old inserter generation. Some parts, conversely, are completely re-designed. Radical design results in significant improvements of functionality or performance, which is important for competitiveness in the market. At the same time, it brings unanticipated problems and issues (Vincenti 1990). Some problems are related to the integration of the control software and the plant. If the integration comes at a later stage of the project, solving these problems might need parts, modules or even concepts to be reworked, which causes delays.

The plant and the controller together deliver the inserter's desired behaviour. It is therefore important to enable their concurrent development. When the control software and plant are designed at the same time, mechanical and software engineers communicate during early development stage, and problems related to plant and control software integration arise early enough to be resolved on time.

The problem is that in the early development stages the plant exists only in sketches and CAD drawings. Interaction with the plant via sensors and actuators is the essence of the control software; therefore, without the plant, control software cannot be tested.

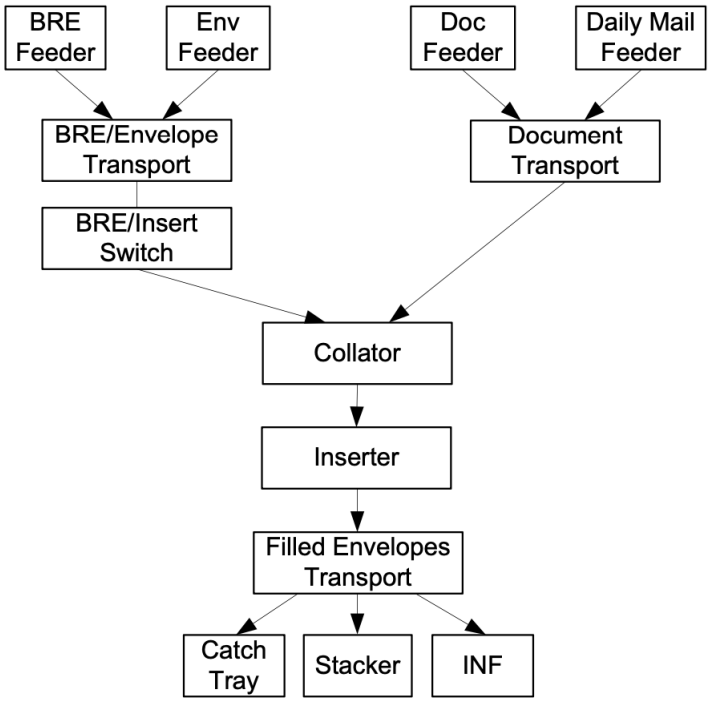


Figure 7.3: Diagram of the mechanical plant parts and document and envelope paths

7.1.4 The Plant model

To enable concurrent engineering of the plant and the control software from early development phases, the company designed the plant simulator (emulator) to test control software. Figure 7.4 shows the comparison of the inserter architecture and the architecture of the testing setup. The emulator ($M1$) is a digital signal model of the plant, sensors and actuators (P). Digital signals represent plant processes, such as document transport and paper folding, and the behaviour of the sensors that detect documents. The signals that emulate plant processes are functions of the signals that the software sends to the actuators. The signals that emulate the sensor behaviour are functions of the signals that emulate plant processes. These functions are specified with LabView (*LabView* 2023) diagrams (Φ) and they define the emulator behaviour. This is a tool that uses state machines as modelling language. In further text, whenever we refer to LabView models, we mean state machine models.

We analysed the LabView model. The programmable hardware (emulator)

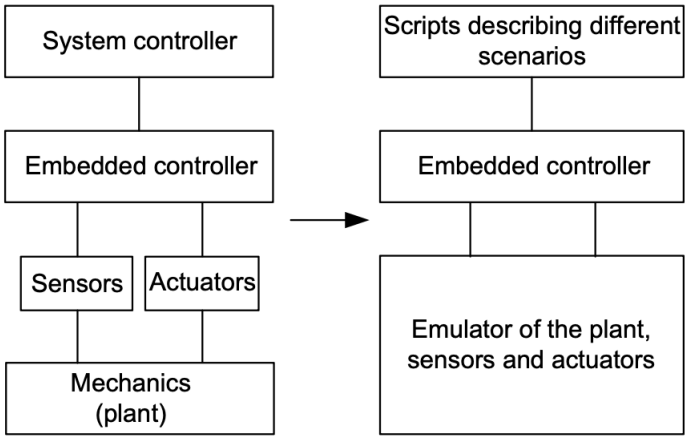


Figure 7.4: Inserter's high level architecture and the concept of the emulator.

is just the execution of the specification, in a way similar to the computer hardware's execution of a computer program. The model could have been just as well implemented as a simulator, describing the plant with software; for our research question this would not make any difference.

Designing the plant for the first time took significant time and effort. The modeller spent a lot of time learning about the plant, trying out different modelling solutions and finding the optimal one. He did not document his insights and knowledge progression, due to time-pressure. The model-based testing improved communication between departments and shortened the integration time, so it will be used in the future.

How the new generation of inserters will look like, when it will be developed, which parts will be incrementally and which radically designed, is not known yet. The modeller will most probably be another software architect either from inside or outside the company. Finally, the tools and languages used now, may not be the choice next time.

The handbook we will present in the next section is written to support modelling of the next inserter generation. It is based on the analysis of the existing inserter model and the estimation of the changes in the next system generation. As such, the handbook is useful for a very concrete case of inserter modelling, or a family of inserters. Possibly, the handbook is useful for a state-based modelling of similar machines, such as printers. But, some of the printers' components and processes are completely different than an inserter's components and maybe modelling them has its own characteristics not covered by the handbook.

Our aim however is not to deliver a universal modelling handbook, but to show how the framework we described in Section 5 can be used to structure the analysis of a model of a highly decomposable mechatronic system. The analysis performed using our framework produces a different modelling handbook

for each concrete case. The handbook can be seen as a specialised modelling method, therefore we could say that we are engineering a method.

7.1.5 The handbook - reusable solutions

The reusable modelling solutions are not parts of the existing model that can be directly reused, but either high-level modelling decisions or solutions represented with state-machines.

Instead of direct model components, we documented state-machines that represent some components of the model. The reasons we did not identify model components for direct reuse was that, in the future, another modelling language might be used. However, the engineers in the company had been accustomed to representing problems with state machines. So, for models, system components that are unlikely to change are transformed into state-machine diagrams. (That is how the solution was designed in the first place - the modeller draws state machines and then transforms them into data flow diagrams of LabView.)

For the components that are more likely to change, we classified modelling decisions by questions that would need to be answered the next time a model was designed and by algorithms of some processes that will not change, even if the physical parts that execute the process change.

High-level system and model decomposition

The inserter simulation model has to follow the system structure. It is often the case that structure of a model follows a modelled system decomposition. If the modelling language allows for hierarchical structures, then the model can follow the system hierarchy and decompositions on different hierarchy levels.

But, how to determine the system decomposition? It is a structure or a number of different structures that the system stakeholders assign to the system. It is the way to deal with complexity when analysing, designing, developing, testing the system. In textbooks, we find examples with neatly organized diagrams representing physical components, functions, processes, etc. In practice, many decompositions can exist at the same time, without having one of them established as *the* decomposition that everyone in the company refers to. The system decompositions in an organization are reflected in the directory tree in the shared project repository, roles assigned to engineers, the different teams they form, and the different abstraction levels they use when working on the system. Often, one diagram mixes different decompositions as well as different abstraction levels.

Take for example the diagram in Figure 7.3. It is one of many diagrams (decompositions) used in communication between engineers in our case company. The feeder refers to a physical component in which papers are initially stored, but when talking about a collator, it is seen as collating function in one occasion, and as a set of mechanical parts in the other. In the same diagram, there is a block called "Transport", which refers to the row of rollers that move documents or envelopes, from the feeder to the collator. But, the rollers that move

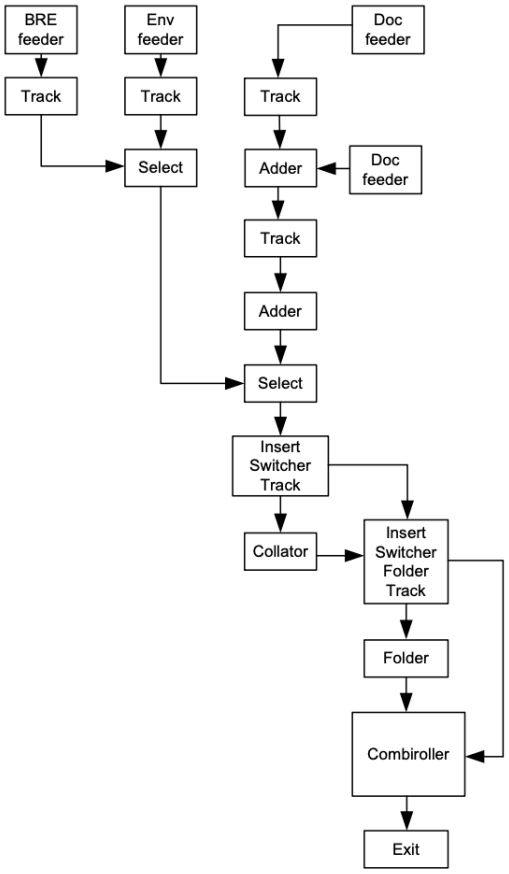


Figure 7.5: Diagram of the mechanical plant parts and document and envelope paths

documents between the collator and inserter, between the folder and inserter etc are considered to be elements of a lower abstraction level and therefore not explicitly represented in the diagram.

The plant modeller had to find a suitable system decomposition to be represented in the model, and it is shown in Figure 7.5. The transport path in this high-level model decomposition stops at the collator. In the actual inserter, the transport path continues all the way to the exit, but in the model its remaining parts are encapsulated in the collator, inserter and folder. How the papers will move within these four components may change in the future, whereas the part of the path, from the feeders to the selector is unlikely to change. Therefore, the high-level structure of the model is the recommended structure for the future models.

Paper transport and paper path

The paper path is the least likely component to change. The current model of the path is designed in LabView, but it is based on the state-machines that the modeller first informally designed and then translated them into the data flow (LabView) model. We chose to document these state-machines (and not the Lab-View diagrams) and present them as reusable modelling solutions, for two reasons. First, LabView may not be the modelling language when the next inserter generation is modelled. Second, the state-based approach is well-adopted in the company and both mechanical and software engineers use them in communication about certain system aspects.

The path (model) consists of the following components:

- Track models a piece of the paper path not longer than the length of the paper.
  - Simple track
  - Track with a slip - models slip of the rollers – rollers are rolling, but the paper is not moving
  - Feeder - this is a special type of a track, that on trigger reads the parameters of the model user. These parameters define the length and thickness of the paper in the feeder.
- Sensor – Every track begins at the spot where the sensor is. The sensor senses the presence of the material in front of it. In the model it can break in two ways – its value can be stuck to represent the presence of the material in front of it; or it can be stuck to represent absence of the material in front of it.
- Adder – Paper moving from one feeder will be joined by the paper coming out from another feeder. The way they join assumes that the software will make sure that they overlap on the two-thirds of their length or more.
- Switch – Models the point on the path where there are two possible ways to take. A switch works like a railway switch – it moves the track towards one path or the other.
- Selector – Selector is the place where a material is coming from one of the two tracks that cross.

Together with every diagram, we provided the vocabulary that defines what each state means, event, action and variable represents. For example, the variable 'length' refers to the length of a single paper or two papers joined together, or it is an envelope length. We also documented a rationale for some of the design decisions and modelling assumptions made while modelling. For example, one of the assumptions is the minimal length of an overlap between two

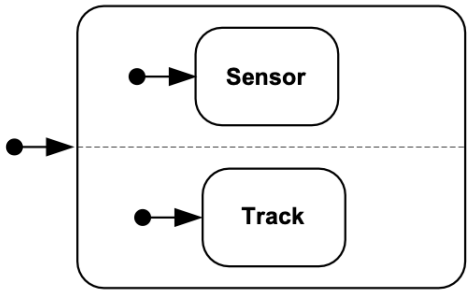


Figure 7.6: A segment of the document path contains a sensor at the beginning of the segment.

A4 papers coming from the two feeders. Some of the diagrams are shown in Figures 7.6, 7.7(a), 7.7(b), 7.8 and 7.9.

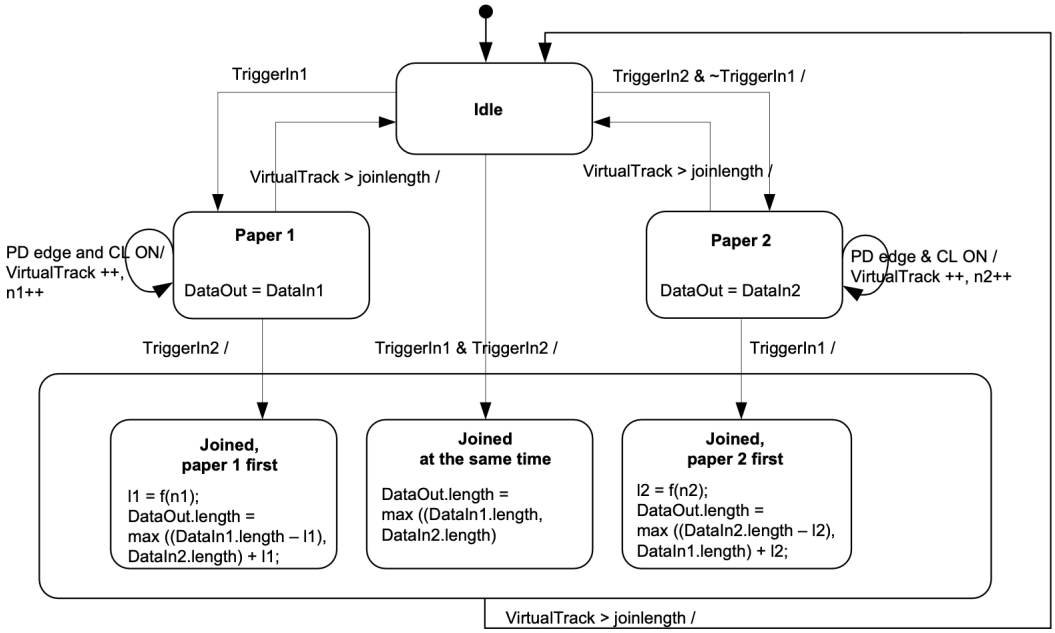
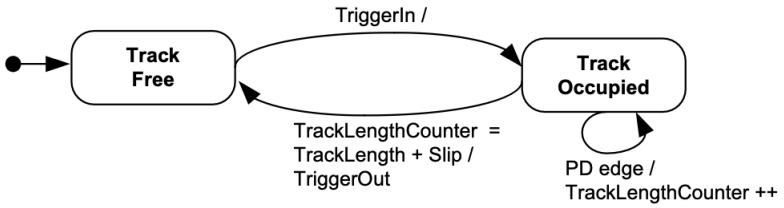
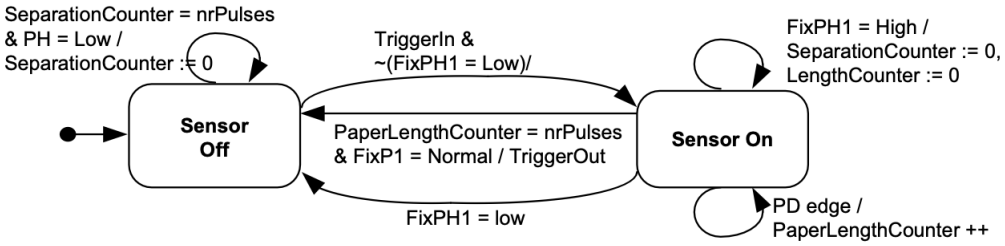


Figure 7.8: Adder represents the part of the document segment path where two documents merge.



(a) Track state machine



(b) Sensor state machine

Figure 7.7: State machines of a sensor and a track combined together to represent a path segment.

Classification of modelling decisions for the rest of the components

The paper path will most likely stay the same, but the collator, folder, and inserter components may change. In fact, it is desirable that some of these components change to make the inserter smaller, faster, or of a different shape. Creative solutions and radical changes introduced by mechanical engineers into new generations of inserters are essential for their competitiveness in the market.

If the physical components change, the high-level processes that they perform will stay the same–inserting papers into envelopes consists of collating, folding, inserting, flap moisturising, regardless the changes of the physical parts that execute these processes. The sub-processes, however, may change, given that the high-level processes are executed by different physical parts. Therefore, we did not generalise the model components that describe the inserter's physical components and sub-processes, but we extracted the questions that the modeller had to answer when modelling them. These questions mark the modeller's search for the solution. They are shown in Figure 7.10.

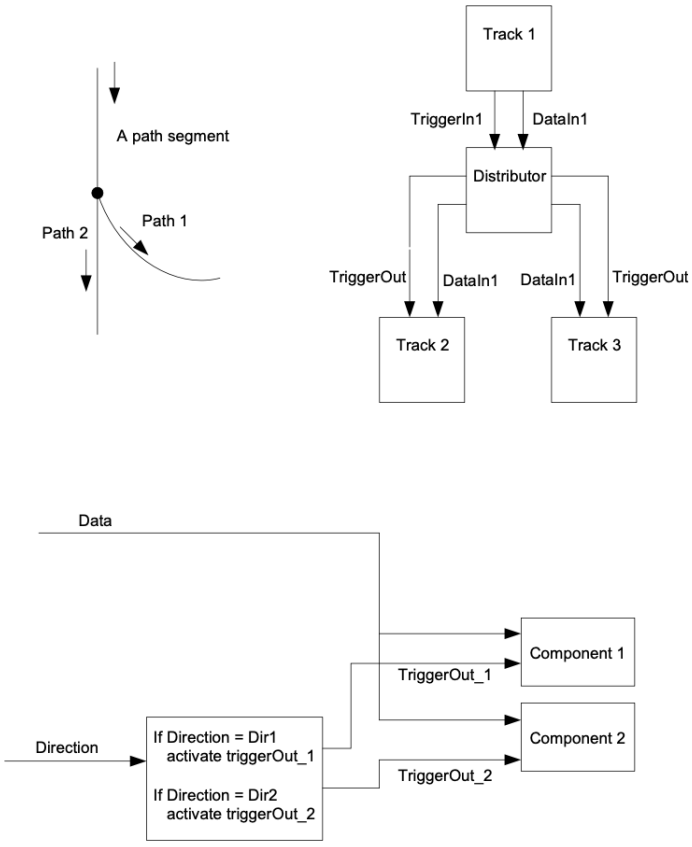


Figure 7.9: A part of the path where it splits into two different paths.

Knowledge about the system

Every state machine came with a short description of the system components and aspects they represents Specifically, we documented knowledge about the documents that were manipulated - papers, envelopes, etc., highlighting which of their properties were relevant (thickness, length, etc).

7.1.6 The Handbook - Reusable Modelling strategies

We documented creative solutions and modelling 'tricks' the modeller invented to prevent the model becoming too complex. The complexity was determined by two factors. The first was the number of hardware circuits that implement the emulator specification. The requirement was to design an emulator that would not require purchase of additional hardware elements. The second factor

Questions to guide learning about new system parts
Functional and physical decomposition What is the system functional decomposition? Which physical part performs which function? Does one mechanical part perform two functions at the same time?
Components and sub-components positions upon arrival and leave of a document Is the initial and end position of a component relevant? If so, what is it? What are other relevant properties of each physical component upon document arrival and leave? How is a document moved from one component to another? Is it a joint activity of two components, or only one of them?
Speeds and speed ratios Is the speed ratio of two mechanical components relevant? Is the component speed relevant and is it constant?
Timing Can certain movements be modelled with abstracting the time the movement takes?
Clutches Is each component moved by a different clutch, or one clutch moves more than one component?
Software dependence Are there sub-functions performed without assistance of the software? What are the relevant phenomena that need to be described, but software cannot observe them?

Figure 7.10: Questions to answer when learning about the new solutions in the inserter design

was the cognitive complexity. It was the modeller's subjective assessment of how complex the model can become and still remain intellectually manageable.

Model elements that do not follow the structure The starting idea was to design a model that follows the structure of the system as closely as possible. This way the design decisions can be justified by the isomorphism between the model and the system. However, sometimes due to the language or other pragmatic constraints, it was not possible to achieve this. We looked for these model elements because they required creativeness to solve problems and to model the system differently. We also recorded the justification for having the component or part of the model not following the 'ideal' structure while still representing the system correctly.

An example of such deviation is the layout of the feeders in the model. The distribution of the feeders in the real system is shown on the left side of the Figure 7.11. The natural way to design the solution would be the structure

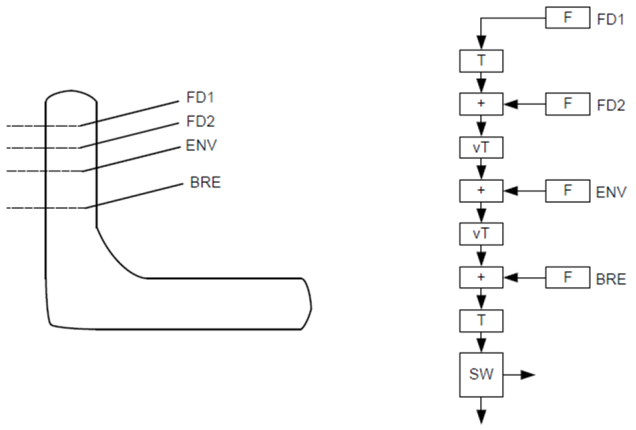


Figure 7.11: Taxonomy of the modelling steps.

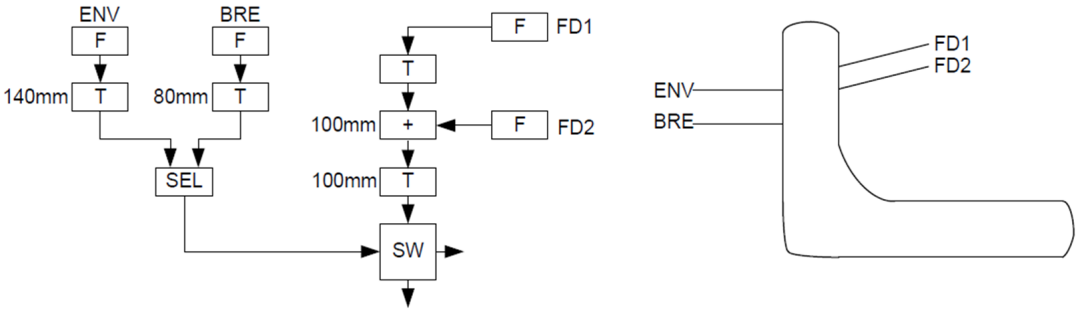


Figure 7.12: Taxonomy of the modelling steps.

shown the right side of the Figure 7.11. This model would follow the layout of the feeders but, due to very short distances between papers, virtual tracks would have to be introduced in the model, so that the movement of each paper can be described. Isomorphism would not be achieved and these virtual tracks would require usage of additional hardware elements.

Instead, the structure shown in Figure 7.12 is used. This structure suggests the layout of the feeders shown on the right side of the Figure 7.12, which is not the structure of the feeders in the modelled inserter. However, given that the overlap between an envelope and a paper sheet is not possible (the control does not allow it), the whole component adequately describes the movement of the documents.

Model elements that appear only in the model and not in the system Some of the model elements do not represent any physical part or a process in the system. Still, they are introduced in the model for various reasons. Sometimes a modelling language has constraints that require adding additional elements, or the system property we are interested in needs to be expressed in terms of the model components that do not have their match in the system. Inventing these solutions also requires creativity.

In the emulator specification, one such element is a sensor. Every segment in the model has a sensor placed at the beginning of the track, whereas in the plant, sensors are not positioned at all these places. But, for modelling the document transport with the desired accuracy, placing 'sensors' at the beginning of each and every segment was necessary to get correct modelling and verification result.

System elements not present in the model A model is an abstraction of a system, which means that many system elements will not be described with the model. For example, in the emulator, the component that moistens envelope flaps is not represented, as this process is controlled mechanically, and therefore invisible to the control software.

A special case of a system element that is not present in the model is the motor. It is present in the model, but 'in person' rather than being modelled. Modelling the motor would require too many hardware elements, so it was cheaper to place the actual motor at the modelling setup. In the modelling setup the motor does not have any load, so its current was scaled down.

Dependence on software behaviour Ideally, a plant model is independent from the software specification, but in practice this would result in a too complex, too big, and incomprehensible model. It is the modeller's decision what assumptions on the software to make to simplify the plant model.

In the case of the emulator, the modeller used the fact that the control forces the feeders' rollers to move the papers in such a way that the papers either overlap at least two-thirds of their length or do not overlap at all. Without these assumptions, the paper path would have to be described with many more elements, and this would require more hardware, and possibly would result in a model too complex to grasp.

Remains of the old model The model we are analysing for future reuse may have evolved over time and it can happen that some of the decisions are based on the system as it used to be in the beginning, but it changed later. If the constraints do not exist any more in the future,

The current model, examined as an example of a good modelling that we want to keep, can still contain some sub-optimal decisions.

Some of the modelling decisions based on the assumptions and requirements that were later dropped off. We recorded them to give the freedom to the modeller to possibly make more optimal solution in the future.

7.1.7 The Handbook - Record of solution Exploration

In searching for a good modelling solution, the modeller tried out some solutions that turned out not to work. To save time next time, we recorded the solution ideas and explanation as to why it did not work. For example, choosing the path segment length with respect to the paper length took many tries until the optimal solution was found.

Also, there were equally good solutions, and the modeller chose one of them. We documented those other solutions, in case it turns out they suit the problem more next time. For example, in the part that represents inserter and folder, some of the elements belong to both modules, so in the model, they could just be represented in either of modules.

7.1.8 The Handbook - Requirements for the model

The model was designed to serve one purpose, but later was used to serve other purposes. We recorded all its purposes, to make sure the model was built for all of them optimally.

Also, we recorded all the pragmatic factors that limited the solution. They influenced modelling decisions significantly, and it might be that in the future, some of these conditions could no longer be present.

For example, the modelling language was chosen for the following reasons: engineers were already familiar with it, it provided visualisation, and graphical modelling kept the focus on the plant (C-like simulation language made software engineers think in terms of the software, not the plant.)

The second factor was limited hardware resources influenced in some sub-optimal solutions. We recorded which were sub-optimal decisions due to this factor.

7.1.9 Model Stakeholders

Assessing a model for future re-use requires understanding of the modelling problem, modelling decisions and the pragmatic constraints on the model. Moreover, it is necessary to get the domain experts' estimation of changes in the future system generations. To gather all the necessary information, the analyst will have to talk to all the model stakeholders.

The table in Figure 7.13 shows the stakeholders of the emulator we analysed. The stakeholders represent roles that also reflect the company's organisational structure.

The system decomposition is assigned within different departments. Mechanical engineers and software engineers use their own system decompositions suitable for the problems they are solving; system engineers use decompositions describing the device on a higher-level of abstraction and may use some of the decompositions of different experts or their own.

Requirements for the model come from both heads of departments as well

Stakeholders	**Goals (G) and Constraints (C)**
Head of System Engineering Dept.	G: Shorten time to market by shortening integration time. G: The model has to be cheap in time, cost and human resources. G: Make modelling faster for future projects. G: Make better models in future projects.
Head of Software Engineering Dept.	G: Shorten time to market by shortening integration time. G: The model has to be cheap in time, cost and human resources; G: Model has to be adequate - to represent the plant so that the results obtained from it are meaningful.
Project leader	G: Finish each phase on time as planned. G: Design the model as soon as possible.
Modeller	G:Make a model of the plant with tools good for the chosen kind of model. C: The plant does not exist yet; C: Modelling some parts would take too many hardware resources.
Software Developers	G: Develop the software and test it. C: Not experienced in language used for programming hardware.
Validation & Verification Engineer	G: Test and verify the software. C: Not experienced in language used for programming hardware.
Mechanical Dept.	G: Deliver the mechanical part. C: Not concerned about the models used for software verification. C: They provide the knowledge about the plant.

Figure 7.13: Stakeholders, their problems and their constraints

as the project leader. Some practical limitations are also coming from software engineers.

The last two groups of the decisions that explain the modelling decisions and document searching for the solution for a modelling problem are coming from the modeller. They are influenced by pragmatic limitations and requirements for the model, but it is the modeller whose experience and creativity determines the modelling solution.

7.1.10 The method validation

In this case, the problem analysis phase was more elaborate, given that we worked on a real case, with multiple stakeholders of the model. This is reflected in an elaborate guidance in the handbook on scoping the modelling problem

Figure 7.14: Problem analysis guidance: a snippet from the handbook

and stating its purpose. In Figure 7.14 we show a snippet from the handbook with the list of questions to answer during the problem analysis phase. (In appendices we show the contents).

During the design of the handbook, in which we captured both reusable modelling solutions, as well as a method (a process) how to design the future models, we worked closely with the modeller in the company to have this handbook in format that is practically usable in the company. Also, as we were designing the handbook we had a number of talks with the main stakeholders of the model and the handbook. Before we delivered the handbook we organised a focus group with seven main stakeholders. Here we went through the handbook and ask for the feedback on the handbook ingredients and format. At that point in time, it was confirmed that the handbook fulfils the given task at the beginning of this case. We also specially interviewed the stakeholders about the usefulness of collecting assumptions, and confirmed that they are not only useful, but crucial to document.

7.2 Summary of the chapter

This chapter describes the industrial case study we performed to validate our method. It was the first "real" system case, after we validated the method on a case in our lab.

What distinguish this case from the previous one is as follows.

- The system is a "real" industrial high-tech system.
- There were already existing informal models of the system structure made by different domain experts
- We dealt with a number of stakeholders
- An aspect of "reusability" of the model has been added - the model is expected to be used for future systems

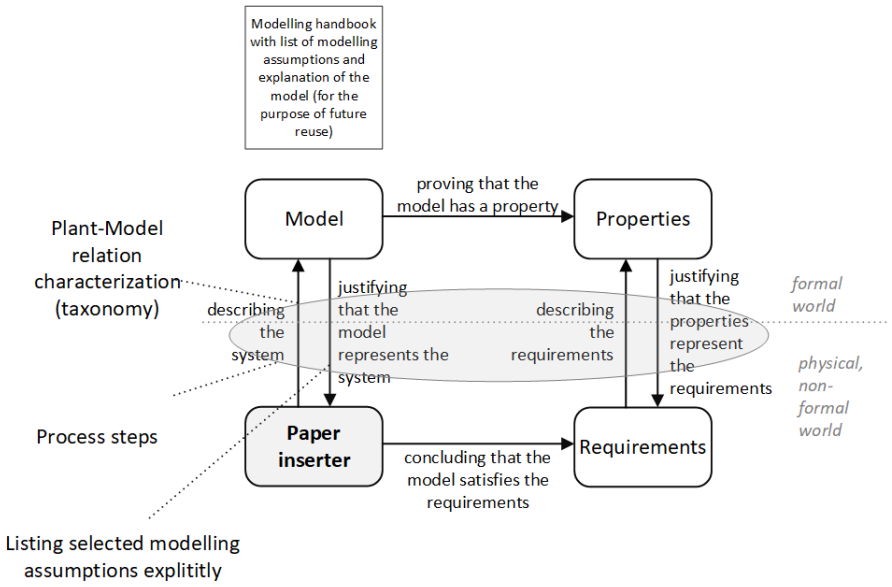


Figure 7.15: Case study: Inserter with the model for current and future use

We started with the method consisting of the following ingredients.

- Taxonomy of modelling steps that provide classification of modelling steps.
- Modelling process steps - the steps describing the modelling process as guidance and checklist when designing the model (DP 1.1) as well as guidance for model justification (DP 2.1).
- List of modelling assumptions explicitly delivered together with the model.

We validated the method in (1) short interviews while working on the handbook and in the (2) focus group we organised at the end of the assignment.

On Figure 7.15 we outline the ingredients of the method after this case. The important of capturing assumptions has been recognised. However, because the domain experts understood state machines, we did not create an "in-between" less formal models. What we did use while exploring the modelling problem and testing the model, we used as extra means of communication with the domain experts small conceptual models they use and that we found in their documentation or gather in whiteboard discussions. This means that in our modelling steps the informal model as in between model is optional depending on what formalisms the domain experts knows. What is added is the use of conceptual models that are mental models, part of collective knowledge in an organisation.

Chapter 8

Model justification: modelling a commercial printer

In this chapter we address design and conceptual problems shown in Table 8.1.

Design problems, conceptual problems and knowledge questions
2.1 Design problem (addressed in **Chapters 5-7 and 8 - 9**) Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model.

Table 8.1: The top level design problem is decomposed into design problems, conceptual problems and knowledge questions.

In this section we describe modelling a commercial printer. The case started with what was then a practical problem/question on usefulness of Uppaal (Larsen et al. 1997) modelling language for different controller models.

Specific about this case study is the following.

- This is our third case study and the second case study performed in an industrial context; the system is an industrial printer.
- The system is more complex than our first case (Lego sorter).
- The domain experts are engineers in the organisation.

- The domain experts do not know the verification formalism and a tool. They are interested in the modelling result.

- In the modelling examples, different ways the users use the printer has been modelled (unlike in previous cases, in which the user in the Domain was using the system in one way only). The system processes are modelled, as well as the Controller.

In this case we applied our method (introduced in Chapter 5) and updated after the initial case (Chapter 6) and our first industrial case (Chapter 7). We applied it to create the model and to justify it. In Figure 8.23 we outline that the research focus in this case is model justification.

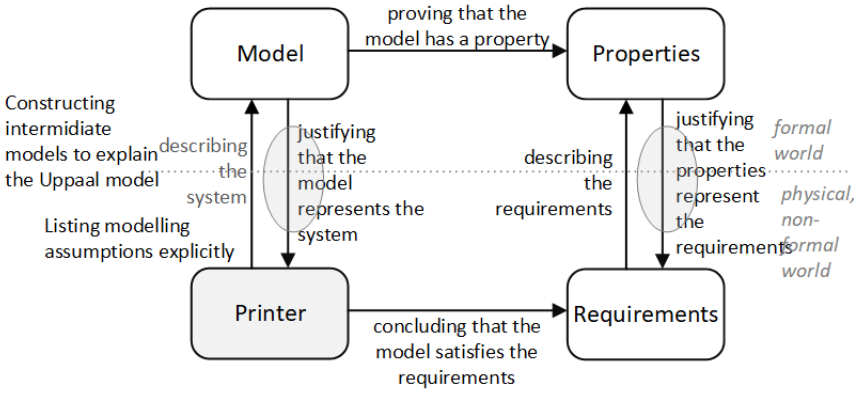


Figure 8.1: Case study: Industrial printer

The result of this chapter is as follows.

- Validated and refined modelling process steps - we recognise that for the explanation of the model, it is worth explaining, and documenting modelling parts of the model that do not represent the Plant or the environment but are there due to the formalism chosen.

- Adding in-between models as part of the model justification.

In the following subsections we describe the practical modelling problem, the model we designed and how we justified it. We conclude this chapter with reflection on the status of our method.

This case study has been performed in Dutch company Océ (*Oce Museum*.)[1]

[1]The company was later acquired by Canon Production Printing (*Canon Production Printing* 2022). The changes in the business and organisation are not relevant for the method presented in this thesis. In the further text we refer to it by using its old name.

8.1 Introduction to the practical problem

8.1.1 The organisation and its business

The company produces printers, copiers and scanners for businesses. Printers are developed in product families. One product family consists of printers for regular office use; they print documents on regular paper, in colour and black & white. Other product families are developed for businesses that offer print services to their customers. Examples of such services are printing booklets, maps, posters, banners, billboards, wall displays. These printers apply ink on various media, including paper and plastic materials.

The company also offers software packages for document management, namely for the following.

- Controlling the printing workflow in a system-of-system environment consisting of multiple printers
- Distributed control over network of the printers situated in different departments of a customer's business (decentralised printers)
- Colour reproduction in display graphics

The business model was at that time, not to sell only the systems and software packages, but to support customers' printing process with a range of services. These services include the following.

- System and software packages maintenance
- Rental and lease of printers
- Selling imaging supplies: inks and toners, different kinds of papers – coated and plain, rolls of paper, plastic sheets).

The Research & Development develops its own basic printing technologies and the majority of its product concepts. There, a few hundred people were employed. As part of the business strategy, the company has innovativeness as one of their core values. Research & Development has established mechanisms (and provides budgets) for collaboration with academia and research institutes.

8.1.2 The System of Interest and Top-Level Requirements

The System of Interest in this case study was the next generation in printer family produced for the regular office use. Figure 8.2 shows one of the earlier generations of such office printer. The top-level qualities (non-functional requirements) for this product family are as follows.

- Reliability of printing (no paper-jam)
- Throughput (high number of papers printed per hour)

Figure 8.2: A commercial printer.

- Ease-of-use (intuitive and ergonomic user interface)
- Image quality: the printer prints the digital image of the document. High quality for the customer means crisp and precise look of the documents.

The marketing department sets the requirements in discussion with the clients, on one hand, and the R & D department on the other. The top qualities, together with other specs are then formulated quantitatively. Reliability can be for example formulated as the number of hours, or percentage of time in which no paper jam, or any other unexpected event happens that requires printer service. The throughput is formulated as the number of papers of specific size, printed per hour. Image quality specification is formulated in dpi and number of lines per inch.

These top-level qualities are achieved by innovative mechanical, electrical and control design. We will explain some of the details in the remainder of this section.

8.1.3 System decompositions

The printer is a cyber-physical system designed by mechanical, electrical, chemical and software engineers. Applying the definition of the Plant and embedded software we introduced in Chapter 2, and showed in Figures 2.3 and 2.4:

- The mechanical, electrical, and chemical parts constitute the Plant.
- The components controlling them constitute the Embedded software. Our focus was the Controller (abstracting away user interface implementation and hardware driver components of the software).

The top-level task of the Controller is to, together with the Plant, print the documents, so that the top quality requirements mentioned above are achieved.

In the Controller, we distinguish the control of the paper movement along the paper path, and additional processes (tasks) performed by the Plant components. The latter are needed to ensure the quality requirements are met. We will refer to these additional tasks as to *maintenance processes*.

The Plant is decomposed into the "warm" and "cold" module, performing warm processes (WP) and cold processes (CP), as shown in the schematic in 8.3.

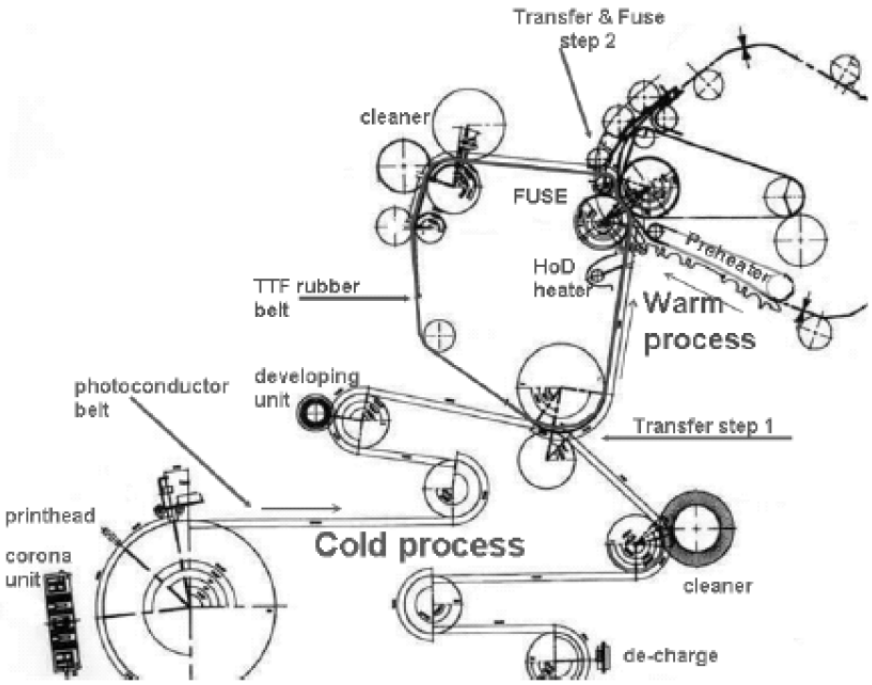


Figure 8.3: Processes and physical components, the picture is copied from (Maurice Heemels 2006).

The warm module maintains temperature higher than the room temperature; the cold module maintains the room temperature. The maintenance processes are performed in both of these modules. The conveyer belt and rollers move the paper along its designated path and and a number of sensors detect the paper.

The Printing process (and consequently its Control) can be decomposed into four main processes. They are as follows.

- Transporting paper sheets.
- Processing printing data.
- Forming toner image.
- Printing.

Paper sheets are moved along the paper path, from the trays to the module where the actual printing takes place. In case of double-sided printing, the sheet is turned over and brought back to the printing module, otherwise it proceeds to the exit tray. After the print request, the printing data is stored into the local

printer memory. It is processed and transformed into a corresponding matrix of electromagnetic signals. The matrix further serves for forming the toner image.

Forming the toner image and printing are performed by two different, mutually connected, modules. The toner image is formed in the **Cold Process (CP) component**, and printing takes place in the **Warm Process (WP) component**. The paper path passes through the WP component.

Besides performing their main functions, the CP and WP component perform a number of supporting processes intertwined with the printing. These processes prevent premature wear-out of the parts and keep them in a state that ensures maximal printout quality.

Each supporting process starts under different conditions, at different points in time. Among conditions that trigger a supporting process are the following events: some of the processes are performed every x number of pages; some processes start if a temperature of a certain part reaches a critical value; some of the maintenance processes can be start by the user. Some of the processes need to be executed as soon as such an event occurs, some of them can be postponed. Furthermore, some of the processes can be interrupted if a print request arrives during their execution, some cannot. The execution times of the processes differ as well, from a couple of seconds to tens of minutes.

The System States. The printer states relevant for the performance (throughput) are the Standby and Run states. The printer is in the Run state while it is printing. If for a certain period of time the printer is not printing and there are not new print requests, the printer control puts the printer into standby state. Some of the supporting processes are performed in the run state and some in standby state. There are also in between states, during which there are also standby processes that can be performed.

Apart from controlling printing, the printer control starts the supporting processes. They are executed in two parallel sequences of processes. This is because the control modules separately control CP and WP component so the processes done by them are independent, with exception of those processes who need to be performed by both modules at the same time. The CP and WP controls have to be in the same states, that is, it cannot happen that the CP control brings the CP component to the standby state, and WP control leaves the WP component in the run state.

The scheduling of the processes affects different printer's performance properties, so a trade off has to be made between the different features. For example, printing can be optimised so that the waiting time for printouts is minimal starting from the moment the first sheet in a batch is printed; alternatively, the waiting time for the printing to start can be optimised. The latter is more convenient for the user in case of printing only one or two pages.

The product generations design Our System of Interest was the next generation in the family of office printers. As it is the case with the high-tech, cyber-physical systems, produced for other businesses - these are very complex and

expensive machines. They evolve, by adding new features to the next product generation, or improving the existing ones. This is incremental improvement, not the radical re-design (Vincenti 1990).

The Controller for the new generation of the printer was largely based on the Controller of the previous printer generation. In this case, the Control designers were looking into how to improve the throughput, by scheduling in a smart way the maintenance processes, intertwined with regular control tasks of moving the paper, printing, and creating the image.

What adds complexity to the problem is the non-determinism of the printing tasks given by the printer users. There can be many tasks sent to the printer at the same time, these can be longer or shorter documents, with arbitrary copies set to be printed. Also, it has not been decided if the priority is given in finishing the current printing task or preventing longer printing time for the potential later print requests.

The goal of industrial stakeholders The question the Control designers had was is related to modelling tools and frameworks that would allow to optimise the Control scheduling, under all the unknowns and non-determinism in the users behaviour. They were exploring different formalisms by testing different tool-suites on a different use cases. One of the tool-suites explored was UPPAAL, and one of the use cases is the modelling problem for this case study.

Modelling problem for this case study Due to the complexity of this problem and limited time we had, we agreed to model two processes, an idealised behaviour of the Controller and model a scenario when the user wants to print a large number of pages.

We worked closely with two stakeholders. One was the domain expert - the software architect (design engineer) of the new printer generation and the other was the owner of the research question (researcher in the software research department). We took an iterative approach and created different Uppaal models with different level of details.

8.2 Modelling method question

The case provided an opportunity to validate our modelling method and use it for the explanation of the model being designed. Our research question is: are the steps of the modelling method sufficient to validate the model?

In this case we modelled the problem, and used the steps of our framework to structure the communication with the domain experts, Control Designer and the holder of the Research question for the company.

After presenting the practical problem solution we will present our findings on these questions.

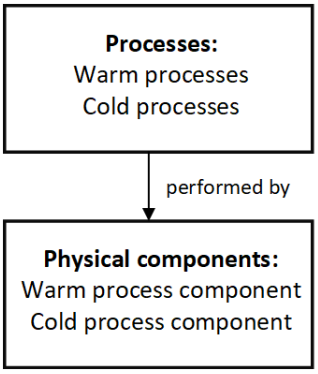


Figure 8.4: Plant decomposed into physical components and processes that these components perform. We modelled the processes.

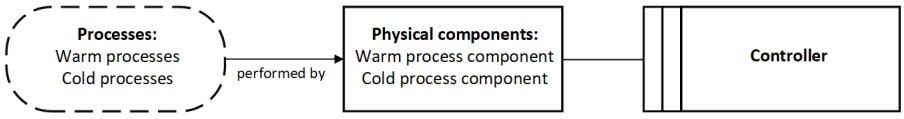


Figure 8.5: Controller with the Plant results in the desired system behaviour with processes being performed in the desired order.

8.3 Practical solution

8.3.1 The modelling task

We modelled the processes of the Plant. For this modelling problem, the physical aspects such as some mechanical properties of rollers or the belt were abstracted away. The distinction between the physical components and the processes is shown in Figure 8.4. In the design document specification only the times necessary for each process to be performed were given, and their order.

In Figure 8.4 the physical components can perform the processes only working together with the Controller. We show this in the problem diagram in Figure 8.5.

The modelling task was to describe the printing process, supporting processes and their different scheduling strategies. The modelling language and tool used was Uppaal (Uppaal 2023). Uppaal is used to describe concurrent processes and formally verify the requirements (properties) and perform simulation.

In our modelling problem, the stakeholders wanted to compare different control strategies for starting and stopping maintenance processes. As the first modelling exercise we modelled one of the strategies and one print request.

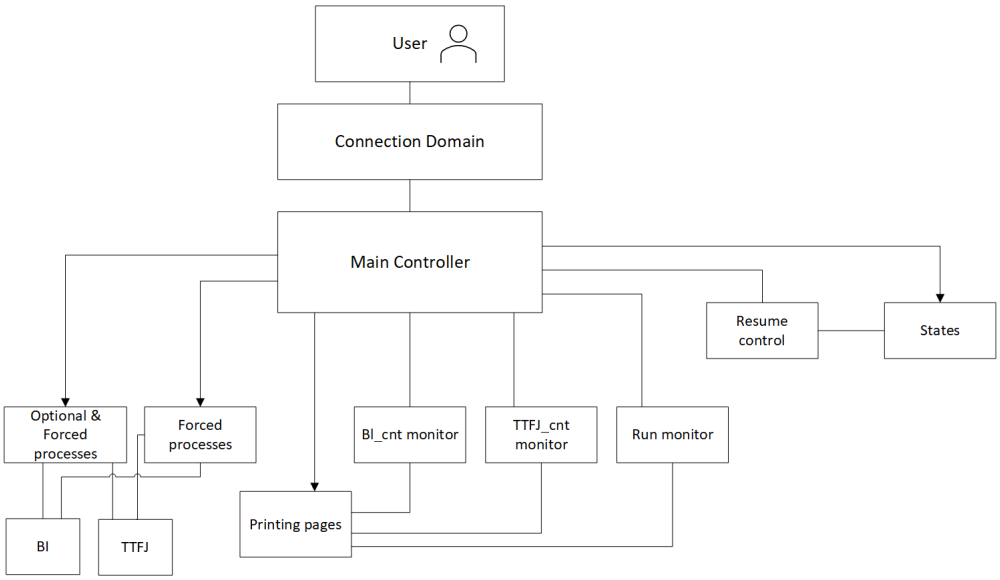


Figure 8.6: 1 - The structure of the Uppaal model representing the Controller.

8.3.2 The solution

Uppaal is a tool that integrates a description language, a simulator and a model-checker. “It is appropriate for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables.” (Uppaal 2023)

High-level structure of the model. The domain experts and the Control designer did not know Uppaal, whereas the Researcher who specified the task did. The Control Designer understood and used state machines. To explain our model we used intermediate, informal diagrams to present the ”architecture” of the model and how it represents the processes in the System.

Figure 8.6 shows the architecture (the decomposition) of our model that describes previously described printing process consisting of paper movement, actual printing and warm and cold maintenance processes. We used this diagram to explain and validate the model. At the same time, being an informal model, it introduced the risk that there is a human error in the interpretation of the formal model. To mitigate this risk, we also used the step-by-step simulation feature of Uppaal.

The first versions of the model had all the control processes in one model. The feedback of the software engineer was that, for a better understandability and extendability of the model, we should divide it into components that relate to main controller and maintenance processes.

On the top level we modelled User and the Connection domain that trans-

lates User requests into inputs for the Main controller. From the Main Controller and below, the processes listed in the design document are modelled. These processes are classified according to the conditions when they can start (whether they can stop the printing in the middle of its task, and under what values of certain environmental parameters and total pages printed) The blocks in this diagram represent Uppaal modelled processes. For each process, there is a Uppaal automaton modelled.

This is the simplified version of what the control can do (it was too complex to model what really happens in the system, and for the purpose of this modelling task, this was enough). The user, via connection domain starts the printing process. The main controller, upon request, starts printing, and counts pages. There are also counters storing the value of pages printed since the last maintenance process. While printing, if the number of total pages printed reaches the value that satisfies the conditions to start one of the maintenance processes, the controller either stops the printing (if the process is forced) and performs the maintenance process, or it continues without stopping.

The meaning of arrows shows the communication direction. Where there is no arrow, the communication goes in both ways. In case of an arrow, for example between the Main Controller and Printing Pages, this means that the Main Controller starts and stops the Printing Pages process, without waiting for any event from the process itself.

There are two additional blocks in this model that do not belong to the process description in the design document. They are the block called "Resume control" and the block called "States". They together with the Main controller, decide which process to activate. They follow events (channels in Uppaal model) coming from maintenance processes to apply the strategy explained below.

In the system, and consequently in the model, not all the processes are active at the same time. We represent this in colouring blocks in the diagram in red. We designed this diagram, after we showed Uppaal models to the software engineer first. It was difficult just by looking at the models, and at the simulation to understand when is what task active. This simple diagram helped. We draw the diagrams as informal ones, there was no tool to automatically derive them, which introduces a possibility of a human error.

- At all times, User, Connection domain, States and Main control are active as shown in Fig. 8.7 - as shown in Fig. 8.8.
- When printing a page, none of the maintenance processes is active. They are in the idle state. While printing, processes that monitor maintenance related parameters are active, as well as related monitor processes are active, as shown in Fig. 8.7.
- When not printing, either OptForced process is running, or Forced process. Examples are shown in Fig. 8.9 and Fig. 8.8.
- When one of the maintenance processes is active, States and Resume control process are active as shown in Fig. 8.9 and Fig. 8.8.

Which processes is optional and which forced is part of the scheduling strategy. Depending on the strategy, maintenance process can stop the current

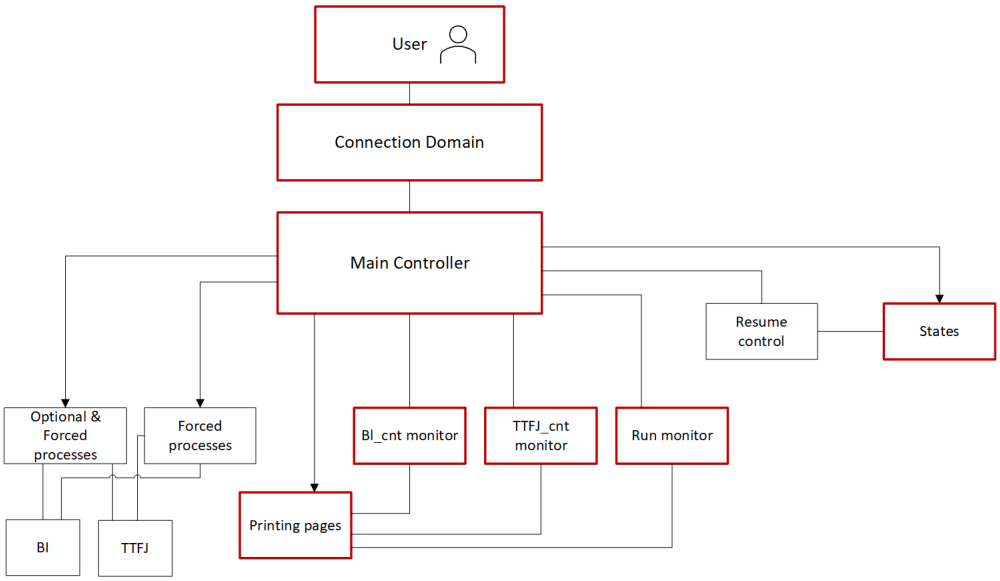


Figure 8.7: 2 - System processes - printing is active.

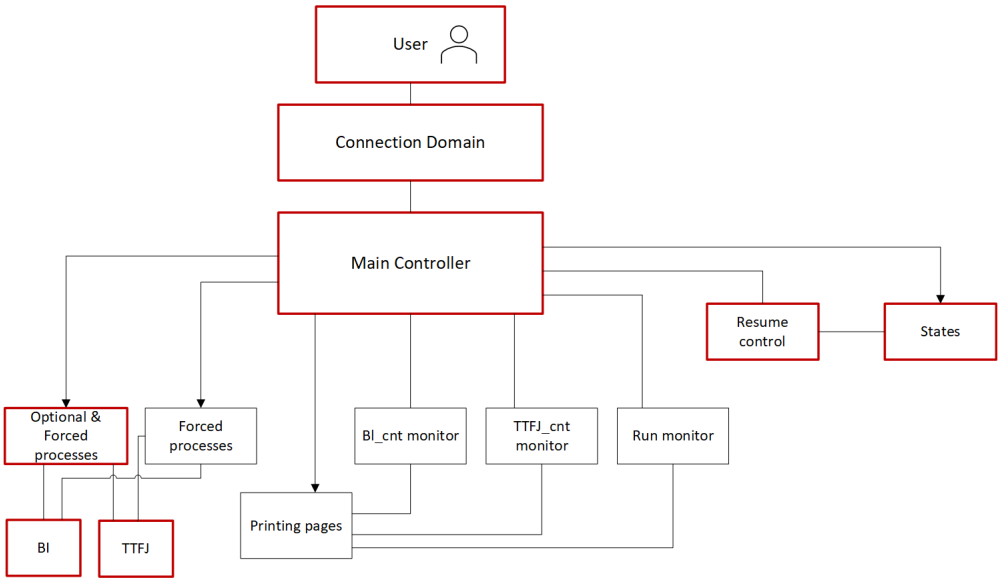


Figure 8.8: 3 System processes - optional is active (as explained above either Forced or Optional&Forced are active, but not at the same time).

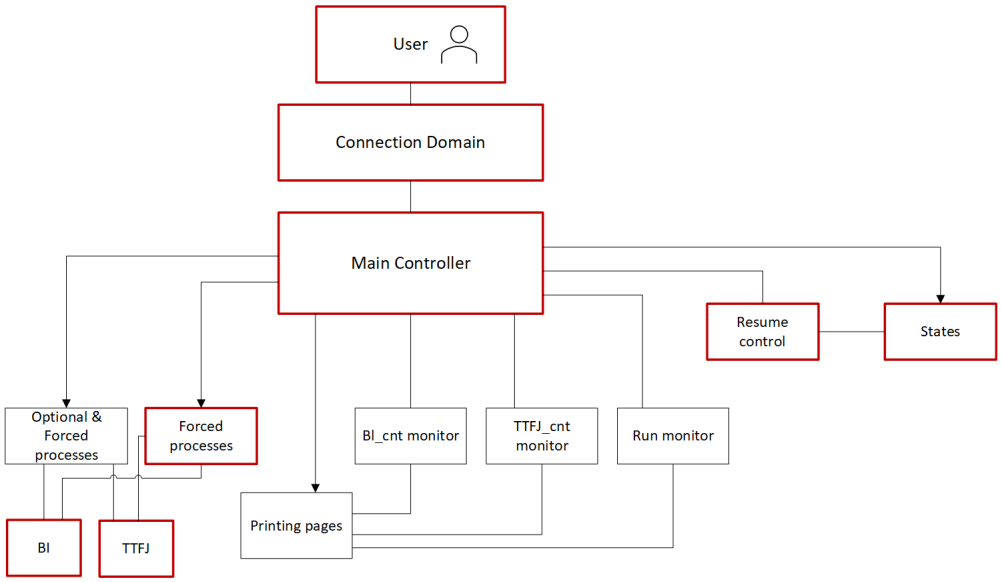


Figure 8.9: 4 System processes - forced is active (as explained above either Forced or Optional&Forced are active, but not at the same time).

printing task, and perform its task. Alternatively, it can wait until the task is finished. In the model. the Scheduling strategies component is a part of the Main Controller, and it is parametrised. The strategy can change, and the rest of the model stays the same. (the printing behaviour, and the maintenance processes themselves do not change, as they represent the Plant behaviour). Which strategy is chosen depends on the parameters of the Scheduling strategy block that is part of the Main Controller. We sketch this in Fig. 8.10. There is relation between scheduling strategy and the trade-off between printing quality properties (throughputs for different scenarios in this case). Due to the complexity of the task and the limited time we did not work on these scheduling strategies. We did manually change some of the parameters for total number of pages to compare the times for finishing printing. We showed these models to the software engineer as a starting point how these models can be used and extended in the future for modelling scheduling strategies.

The following pseudo language statements describes how printing behaves when a forced and optional process are due. In the following statements "run" means printing all the papers of one task. The statement "page is printed" means printing one page has completed. Printing one page always finishes. The maintenance processes cannot start in the middle of the process of printing a page.

If (a page is printed) AND (the run is not finished) AND (nothing forced) (resume printing)

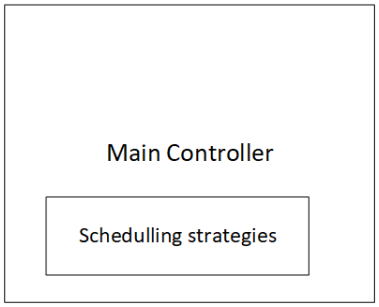


Figure 8.10: 5 The Main Controller contains a selector of the scheduling strategy.

If (a page is printed) AND (the run is not finished) AND (something forced) (do only forced things) (resume printing)

If (a page is printed) AND (the run is finished) (do optional and forced) (stay in StdBy) until (printRequest)

A reflection on the intermediary diagrams The diagrams we used to explain the Uppaal model took time to draw and took time to explain properly to the domain expert. It would have been ideal to use instead a model designed using some Domain Specific Language (DSL) with a good visualisation. In fact, these kind of models are being developed in the company as support for the design and verification process, but at the time of performing this case study they were not ready.

Uppaal models

Here, we present Uppaal models that implement the processes shown on informal diagrams in previous subsection, in Fig.8.6-Fig.8.10.

We modelled the use case in which the user finish the last page of the previous task, wait and then start the next task. The Connection Domain translates user requests to input signals for the Main Controller, and output signals of the Main Controller to the User.

The tested if there was no deadlock, and for different values of the parameters (number of pages) we manually looked for the time via successive approximations ($E <> cnt\_total\_exec > xxxx$, where xxxx is the number of time units).

8.3.3 Model justification

To test the model, to validate it, we performed simulations available in Uppaal toolset. Then we showed the model and the diagrams to the domain expert and discussed it. As explained above, we used the diagrams of the model architecture and the Uppaal model simulator to explain the model to the domain expert. What turned to be extremely difficult and required multiple rounds of reviewing the models was that the domain expert provided lots of additional,

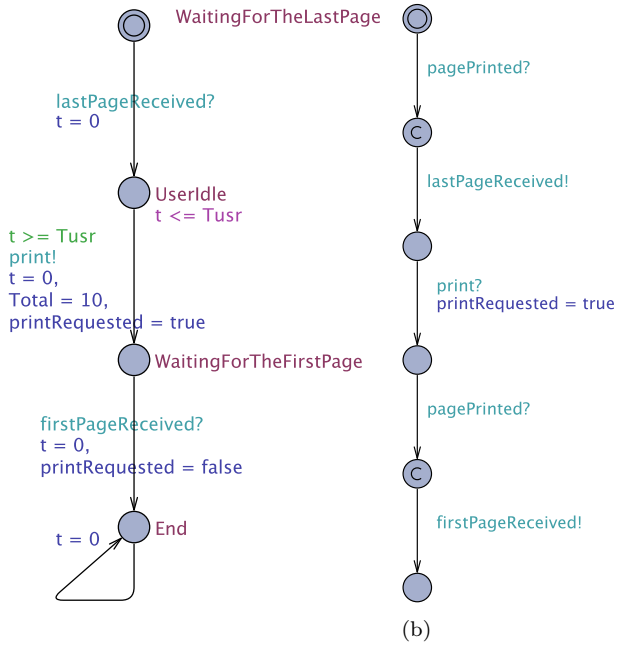


Figure 8.11: (a) User and (b) Connection Domain corresponding to the User and Connection Domain blocks in Fig. 8.6 - Fig. 8.8. In Uppaal, the location with C is used to create atomic sequence. In the model Tusr represents the time the user does not do anything before they start a new printing task.

software implementation related details that were not present in our model. We abstracted them away to first test the strategies. We used abstractions to test strategies and the challenge was to explain why with these abstractions the model represents correctly the strategies.

8.3.4 Validation Steps of our method

Problem analysis

The validation of the modeller's steps during knowledge increase goes into two directions. First, it ensures that the modeller understood the problem correctly and that the stakeholders gave correct information about the system and formulated the problem correctly. By understanding the problem we mean the following: (1) understanding how the system works, (2) understanding the system requirements, and (3) understanding the requirements for the model itself.

To discover how the system works, we used the existing documentation and talked to domain experts. We explored (1) how the control works, (2) how individual components shift from one state to another and (3) how processes and

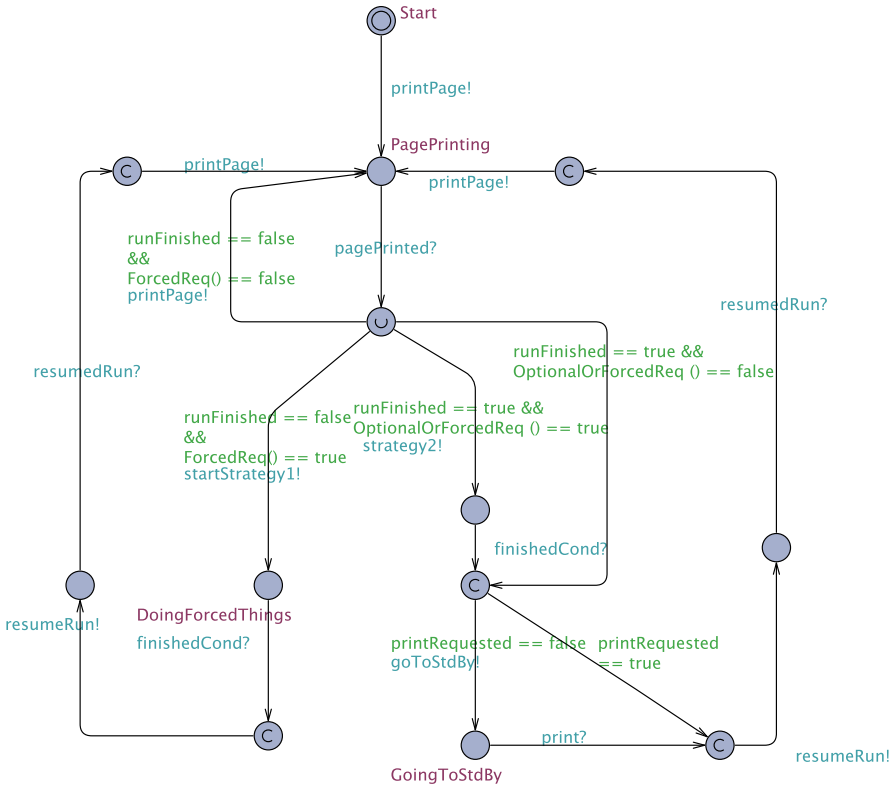


Figure 8.12: The Main Controller corresponding to the Main Controller block in Fig. 8.6 - Fig. 8.8

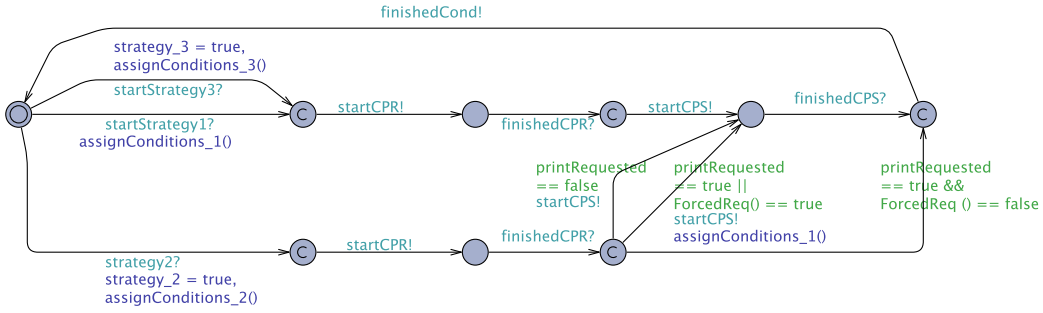


Figure 8.13: Scheduling strategies determining The Main Controller behaviour sketched in Fig. 8.10

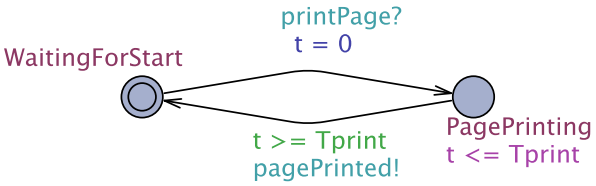


Figure 8.14: Printing a page corresponding to the Printing pages block in Fig. 8.6 - Fig. 8.8.

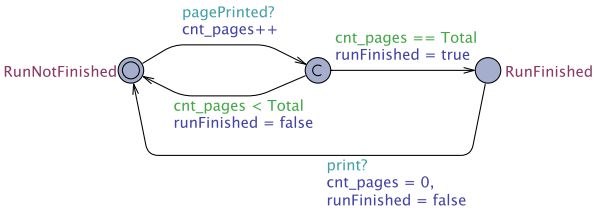


Figure 8.15: Run monitor process corresponding to the Run monitor block in Fig. 8.6 - Fig. 8.8.

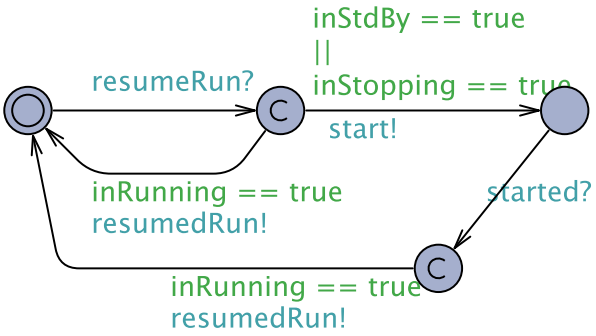


Figure 8.16: Resuming Running process corresponding to the Resume Control block in Fig. 8.6 - Fig. 8.8

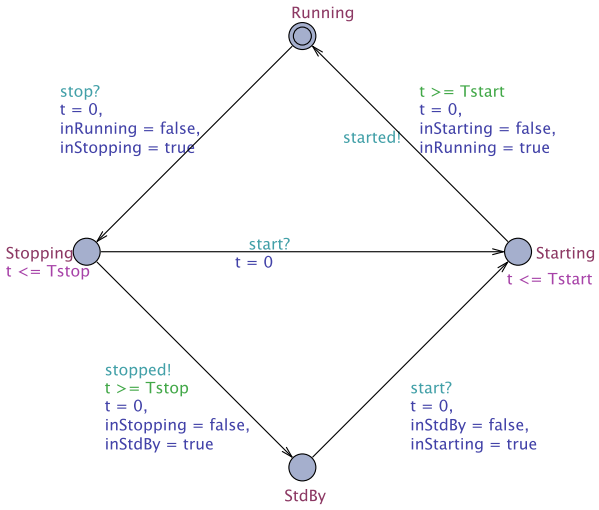


Figure 8.17: Machine states process describing the current machine state, and corresponding to the States block in Fig. 8.6 - Fig. 8.8.

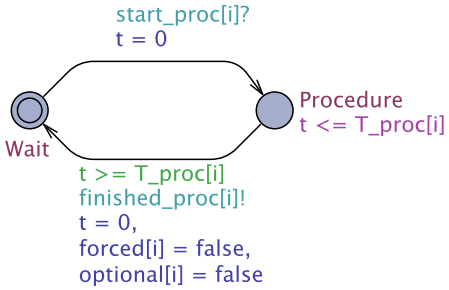


Figure 8.18: Template for parallel conditioning processes (TTFJ and BI process models are instantiated from this template and correspond to corresponding to the TTFJ and BI blocks in Fig. 8.6 - Fig. 8.8).

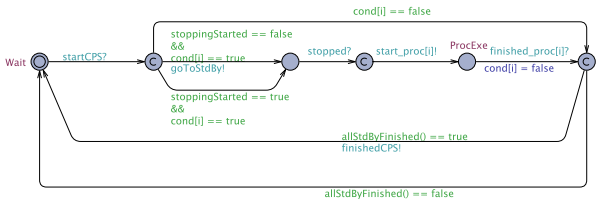


Figure 8.19: Part of the controller (subcontroller).

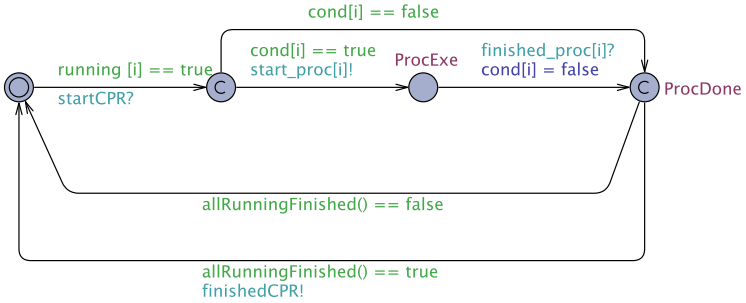


Figure 8.20: Part of the Main controller (subcontroller).

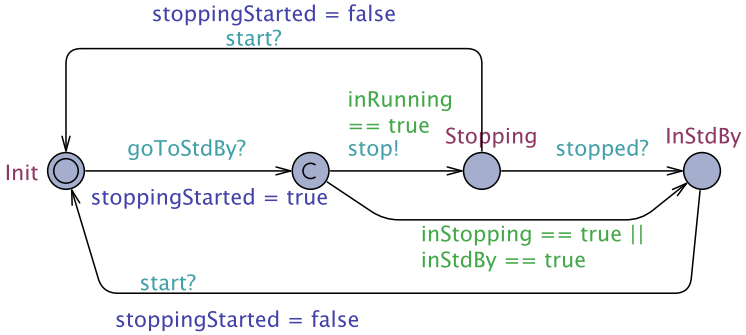


Figure 8.21: Part of the Main controller (subcontroller).

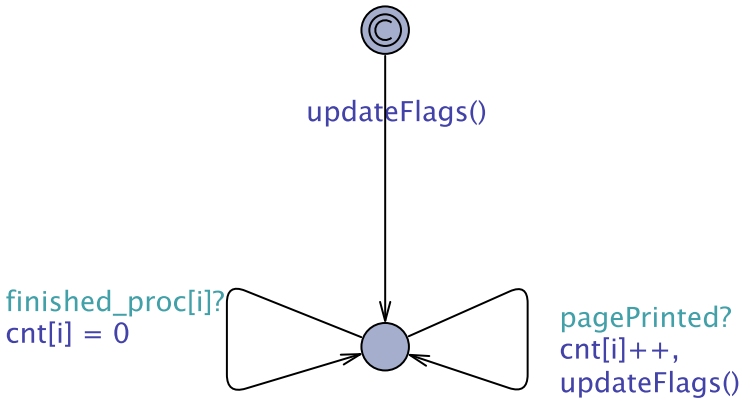


Figure 8.22: Template for counters. This is auxiliary part of the Uppaal model not corresponding to any of the processes. It is there to enable the counter for units representing timing units.

plant parts work. Our discovery was related to finding the right abstraction level to represent these components, for both the problem-owner and the modeller.

The problem owner gave very detailed descriptions of some components and more abstract description of the others. The sources of model errors in the first versions of the model were: (1) the modeller did not understood correctly the documentation or the sketches of the domain experts; and (2) the problem-owner gave too detailed or too simplified explanations of some components – for the purpose of the model. As an example of (1): the documentation describes states of the printer and states of the components with the same name: active, standby etc. It took some iterations to understand which component reaches which state when, and how this relates to the states of the printer. An example of (2) is that the goal of the modelling was to prototype a model for certain situation before modelling all possible maintenance processes. This prototype situation required abstraction along another axis, in an iterative steps of exploring what the tool suite and the formalism can do.

An aspect of the problem validation is requirements elicitation. In this assignment, the main goal was to check how already a satisfactory performance can be improved. Modelling scenarios forced the problem-owner (the software architect) to be more precise, and brought insights and structure into their own perspective of the problem. This is inherent to requirements elicitation. As we already mentioned, we started from a number of simpler cases to come to an illustrative prototype. It took some iterations to come up with what illustrative case is.

Another set of requirements were those for the model understandability and usability. For example, we simplified the description of the User (behaviour), with an equivalent model that led to a valid verification result. It is about the user starting the next printing task after finishing the previous one. As the model is about the current task, finishing the last page of the previous task is not necessary to model. However, the model stakeholder found this simplification too "far away" from the real process. So we made the model slightly more detailed, so that in the Uppaal simulation environment they would be able to also simulate the User finishing the previous printing task and waiting before starting the new printing task.

Also, in relation to modelling the User behaviour, we modelled "connection domain" and in the validation of this part, many assumptions on the user behaviour are captured.

Model design

When constructing and explaining the model, we used our taxonomy (shown in Figure 5.3. One of the main abstraction was to model the processes and not the physical components, and to model the Controller of the processes, abstracting away the controller of the physical components.

The decomposition to the Plant processes was given in the design document. We followed that same decomposition in the Uppaal model. We do not have the full isomorphism between the Plant decomposition and the Uppaal model

decomposition, because the document described the desired Plant behaviour without the decomposition to the controller and controlled parts. In our model we designed the controller controlling the processes and added some new processes that track the states and count time.

We validated the model by designing queries that explored different safety and liveness model properties. In our model, the most important thing was to check whether all components are in a correct states, given that this was not obvious for some border conditions. We also applied model-checking to validate correct order of processes execution.

8.3.5 Model justification

Model validation (for the modeller) Uppaal has a simulation environment, which we used to validate the model. We also used this engine when explaining the model to the stakeholders.

Explaining the model to the stakeholder As already previously mentioned, we designed intermediate models to explain the Uppaal model structure. This step enhanced the model understandability, but as it is the case with informal models, they introduce the risk of being not correct. Our diagrams were in a way an informal DSL (domain specific language) that showed the domain expert the structure of the control. What is missing here is that this is not a real DSL, and there is no automatic translation of the diagram to the Uppaal model. The translation from the diagram to Uppaal model and vice versa increases the risk that the model is not correct.

The diagrams were used to explain the model structure consisting of a number of automata that relate to processes of the Controller and the Plant. The behaviour of each automata is described in Uppaal formalism and language, that has a lot of commonalities with state charts, but also a number of differences. Even though we explained step-by-step the model, it is not easy for someone who has not been using formal methods earlier to grasp the concept of urgent channels and other elements of Uppaal, that are not present in state charts. To minimise the risk of wrongly understanding the model, we also developed additional test queries that addressed phenomena described with urgent channels and locations.

Collecting assumptions We already mentioned that verification results hold under additional assumptions. We collected these assumptions and checked if they are fulfilled when printing is in the states we modelled. We delivered the list of assumptions together with the model. This list is never complete, but some of the assumptions are important to document, otherwise they can be easily overlooked.

We presented them to the domain experts in order to check their validity. We also classified them in order to check with the domain experts if there are more assumptions that we overlooked. Some of these assumptions always

hold, and some are the idealisations that we introduced. Their role is to (1) document conditions under the modelling results are correct and (2) to test with the model-stakeholders if the idealisations we made still result in an adequate model.

Following assumptions and classes are just some of those we collected, given as illustration.

- Decomposition to: User, Environment, Plant, Paper, Control. Here for example, we documented an assumptions that during printing, the user will not try to open the printer door (this refers to the printer behaviour). Also, we documented that the time to forward the print request through the network is negligible. The first assumption is the condition under which the user can expect the performance that is promised by the producer. The latter assumptions is more an idealisation, but it should be explored what happens if there is a delay.
- Decomposition to processes performed on paper: printing, transporting. Here, one of the assumptions recorded was that printing takes 0.5 seconds. This is the assumption under which the result of the verification hold. It can happen that under some circumstances we need to take into consideration longer page printing times.
- Functional decomposition - here we focused on the assumptions we took about individual processes like:"Component X will never become too hot."
- Errors and faults - one of the assumption here is "Paper jam will not happen".
- Initialisation and finalisation concerns - here we addressed initial values of different counters.

8.4 Conclusion of the case

In this case we designed and verify a model of the Controller for two Printer processes. Our focus was on justifying the model's adequacy.

Performing the modelling task in a complex industrial setting, means first of all that the question:"What is the model purpose?" have to keep being revisited along the model design.

Another lesson learned from our case study is, that when it comes to increasing the stakeholders' confidence in the model, it can be a negotiation process. For complex industrial systems, one model does not cover all the aspects and the requirements, and for some analysis, a model that is not 'perfect' may be good enough.

Finally, there is a danger in explaining the model to the stakeholders that are not experts in the language and the methods that the modeller used. In our example, Uppaal diagrams look very much like some of the state diagrams the stakeholders used, but there are semantic differences that can change the complete meaning of the model.

The answer to the question of the industrial counterpart about using Uppaal for finding a scheduling strategy for many scenarios of users' print request was

that Uppaal framework is not suitable. So we agreed to perform modelling for 2 typical scenarios and show the simulation in the simulation tool of Uppaal.

We explained the Uppaal model using high-level diagrams and explaining how we translated the control specs in English to formal models. For our research questions the focus was on how to explain the models and on building the justification argument that the model is adequate. The results of this case study are sketched on Figure **??**. We used the conceptual framework for modelling, as well as explaining the model. We also recognised the need to explain the model using intermediate, less precise models. These models add to the justification argument, and improve understandability of the models. Their downside is that they are not formal which brings the risk of human error in the explanation. We also confirmed the importance of documenting modelling assumptions.

In this study, there were no new elements added to the method.

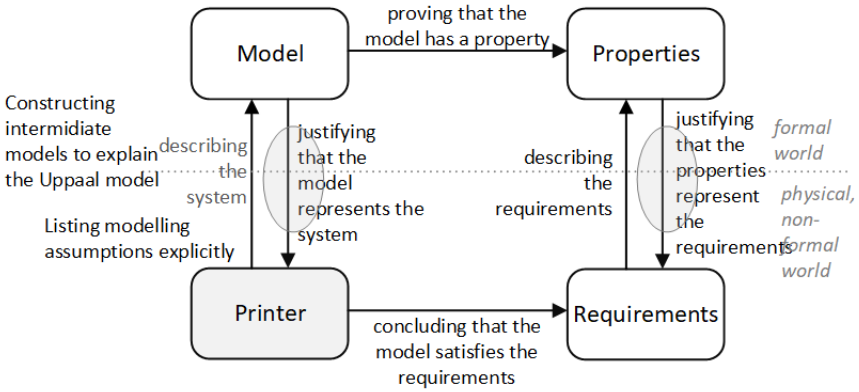


Figure 8.23: Case study: Industrial printer

Chapter 9

Modelling assumptions

In this Chapter, we address design and conceptual problems shown in Table 9.1 (this table contains a subset of questions presented in Table 1.1).

Design problems, conceptual problems and knowledge questions
2.1 Design problem Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model. **2.1.1 Design sub-problem** How to organise justifications to facilitate communication among different domain experts?

Table 9.1: The top level design problem is decomposed into design problems, conceptual problems and knowledge questions.

The result of this Chapter is an extension of the design problem to design a justification argument with the technique and guidelines to capture the modelling assumptions. At the same time, the steps proposed in the guidelines answer the conceptual question of what constitutes a justification argument.

Figure 9.1 sketches the scope of this Chapter.

The result of this Chapter is as follows.

- Further elaboration of the modelling assumptions to collect in communication with control engineers, while co-modelling.
- Further extension of the method to collect modelling assumptions.

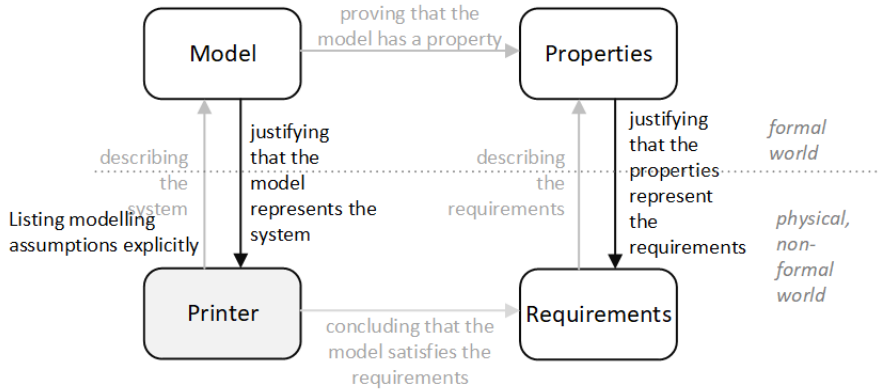


Figure 9.1: Modelling assumptions as part of the justification argument

In this Chapter we will reflect on the assumptions collected in cases presented in Chapters 6, 7 and 8. Additionally, we look at another case performed within a research project that designed co-modelling method and tools; there, many assumptions were included in the method.

9.1 Introduction

In previous chapters, we identified and structured modelling decisions, which answers the Design Problem 1.1. The outcome of these decisions is the model itself – its elements, its structure, and its relations to other models. In Chapter 5, we also introduced the correctness argument, and extended it with the notion of assumptions in Chapter 6.

The correctness argument holds only if the model is valid, that is, if the model correctly represents the system with respect to the given requirements. Therefore, one of the modelling steps is to establish the model's validity (correctness).

Given the informal nature of modelling, it is not possible to prove mathematically that the model is valid. Modelling must cross the gap between the problem domain of the system and the mathematical world of the model, and the modeller can only increase the confidence in the model by (1) making experiments and testing the model itself and (2) by talking to the domain experts, and reading the documents. Often the system is incrementally changed, improved, extended, a new feature is added. The modeller can in this case review the existing code (which is not always helpful and depends on how readable the code is) and if they exist, the models of the older versions of the system.

After the system is built, it will be tested, and the tests may uncover inconsistencies or incorrectness of the requirements, bugs in the models, or bugs in the software implementation. By then, the costs of the redesign are higher, as they grow together with the timeline of the design and implementation process.

In an ideal case, the model will be verified and validated, and the bugs in the control design will be eliminated before the actual system is built.

While exploring the solution space, the modeller will unavoidably make a number of assumptions about the system, the requirements and about the problem domain in general, that is all that is in the controller environment. Examples of the assumptions are:

> "The sensor's accuracy does not depend on the machine's temperature."
>
> "Every piece of luggage placed on the luggage handler does not exceed 50 kg."
>
> "In the document, parameters *new_wafer* and *wafer_new* refer to the same condition."
>
> "Another software sub-system will not send the *stop* signal before sending the *start* signal."

These are all the assumptions made about the controller environment. Additionally, the controller model itself may consist of different sub-models which are based on two different formalisms, one being the control engineering framework and formalism and the other software engineering one. For example the software modeller may assume that the angle direction in the control model increases in the clock-wise direction, whereas it may be the opposite, and that the angle increases in the anticlockwise direction.

If these assumptions are not fulfilled, the model correctness, and consequently, the system correctness, cannot be guaranteed. Assumptions are therefore an important element of the model and system design validation, and in this chapter we extend our conceptual framework with the explicit notion of modelling assumptions. Checking the correctness of the model means making its assumptions explicit and checking if they are true of the system at hand.

9.2 Assumptions in the modelling process

9.2.1 Definition

According to *Merriam-Webster dictionary* (2018), an assumption is *"a fact or a statement (such as a proposition, axiom, postulate or notion) taken for granted"*. In other words, it is treated for the sake of a given discussion as if it were known to be true. This is, when they are "treated" - brought up into the discussion. We also have many situations in which assumptions stay "untreated", meaning that no one is aware of having made the assumption.

In the context of embedded system modelling, we define **modelling assumptions** as postulates about the environment of the control software being modelled, that are (1) relevant for the model validity and (2) not described in the model.

The control software being modelled may be a small part of a complex software system that steers a complex plant. Therefore, its relevant environment

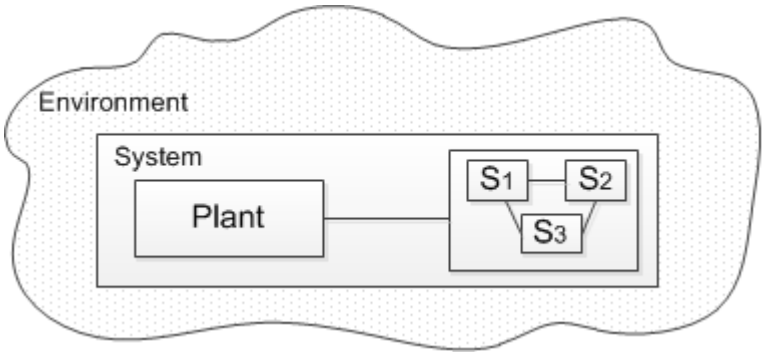


Figure 9.2: The environment of the modelled software sub-system *S1* consists of other software systems *S2, S3*, the Plant and the overall System's environment.

may consist of not only the plant, but other (software) sub-systems, as illustrated in Fig. 9.2.

Our definition emphasises that the assumptions are the conditions that are *not* described in the model (otherwise they would not be assumptions). In the example in the introduction of this chapter, we mention a constraint that the *stop* signal will not arrive before the *start* signal. Let's assume that the modelled component is that of a printer, and that it is modelled following component-based design that distinguishes between component internal behaviour and the behaviour on its interface that with the other components.

A software component C and its interface Ci are shown in Fig. 9.3. The component C controls printing tasks on a printer. It receives the print request from another component that handles requests at the user interface. Component C receives these requests through its interface Ci. Concretely, the interface Ci receives *start* and *stop* requests and handles them. The component C behaviour model describes/specifies the internal component behaviour upon receiving start and stop signals at its interface. Upon receiving the *start* signal, the component reads image data from another component, it checks if the printer has finished previous printing task, it actuates motors and so on. It coordinates other components that are controlling the printing behaviour. Upon receiving the *stop* signal, it interrupts the printing by sending the paper to the bin, stopping the motors of the conveyor belt; it sends the message to the user interface.

Following the philosophy of component-based design (Szyperski 2002), the interface (model) hides internal component behaviour and shows the protocol, the hand-shake between the interface Ci and the other software component that passes print requests. The communication protocol on the interface is described with a simple Uppaal model and shown in Fig. 9.3 as well. The protocol starts in the *Idle* state. After the *start* event, the interface, and consequently the component C changes its state into *Active*. In this state C will perform some actions until it receives a *stop* event on its interface.

Looking at the model of the interface protocol, we observe that an implicit

assumption has been made on the order of signals coming. The assumption is that another component will never initiate a *stop* event before the *start* event. Another assumption being made is that the *start* event will arrive only once. What if we want to reinitiate the behaviour by sending the *start* signal multpile times? The interface protocol model does not describe this possibility. In the system though, it can happen that another component communicating with the component C can send *start* signal multiple times.

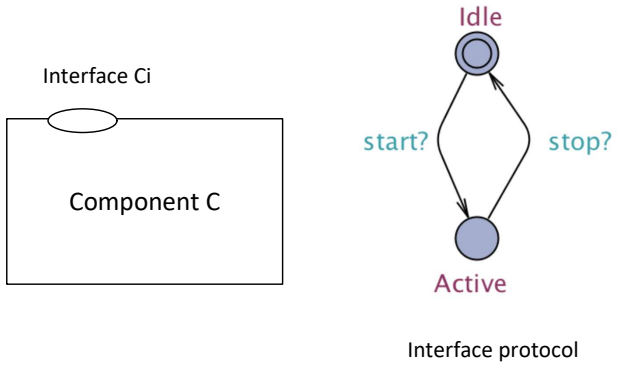


Figure 9.3: The assumptions made: (1) Another component will not initiate a *stop* event before firing a *start* event; (2) Another component will not fire a *start* event multiple times in a row.

The designer may decide to extend the model of the interface protocol by adding the possibility of the *start* signal arriving multiple times. That communication protocol is shown in Fig. 9.4. After this possibility is formally described, it can be verified and validated. For example it can be verified that multiple start signals will not send the component into a deadlock or incorrect behaviour. And it can be validated that this is what the software and system engineers wanted (and not for example issuing error message on multiple start signals arriving after each other). Therefore it is not an assumption any more, but a confirmed system property. The belief that the *stop* event will not arrive before the first *start* event is still an assumption.

Apart from the assumptions on the environment, plant and software and control specifications, there are many other assumptions that we do not take into account. For example the model's implementation in the code and the hardware of the embedded system are out of our scope. This does not mean

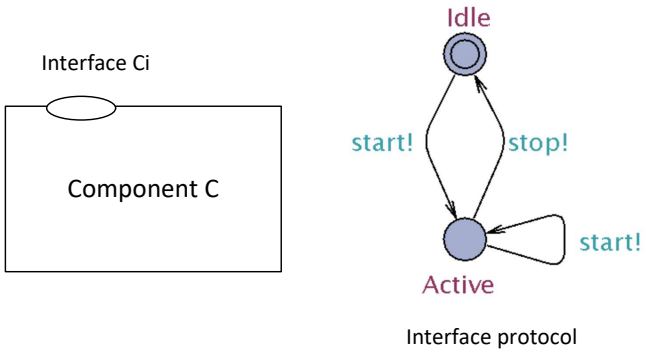


Figure 9.4: The remaining assumption is that (1) Another component will not initiate a *stop* event before firing a *start* event.

that they are any less relevant or important, only they refer to the system parts that are out of the scope of our analysis.

Modelling assumptions are made while designing the model and they are relevant for the model validity. The modeller is aware of some of them, and some of them they make unknowingly. Because their fulfilment guarantees model validity, they are automatically part of all of the system assumptions relevant for the system validity.

9.2.2 Problem statement

There are a number of problems related to addressing assumptions to increase the confidence in the model. The first problem is that the assumptions, not being part of the model, cannot be formally verified. Their fulfilment can only be informally confirmed. The second problem is that the assumptions often stay implicit. They are often unknowingly made. The third problem is that different groups of designers make mutually incompatible assumptions, which leads to errors later on, during system design, implementation, or use. Or, they miss relevant assumptions that they are unaware of.

The relevance of an assumption may change over time. What is irrelevant today may become relevant tomorrow. Since we do not know what tomorrow looks like, we do not know what will become relevant tomorrow. If something in the environment changes and leads to an unwanted emergent system property

or a behaviour, the model needs to be revisited and the system redesigned to eliminate it.

If a selected number of the assumptions are collected and written down in a document, the designers can check their compatibility and confirm or dispute whether they are true or not, at least in a present moment. (There is also the question if the assumption is necessary, even if it is true.) The problem then is that there is an infinite number of assumptions designers can make, implicitly or explicitly, and it is up to the designer to decide which subset is a reasonable choice to deal with and document explicitly. For example, for a nuclear power-plant, an assumption that an earthquake of a certain strength may be followed by a tsunami of a certain power, has been a crucial assumption for a nuclear power-plant in a seismically active geographic area close to the ocean. But, in other cases, documenting and validating such an assumption would become a burden, an overload of an already extensive documentation.

In the remainder of this chapter, we address the challenge of systematically documenting system postulates and constraints (that is, assumptions) that are not formally described in the model. This may additionally help uncovering the assumptions that would otherwise remain implicit and choosing those that are relevant. Deciding what assumptions are relevant, analysing them to find possible inconsistencies, and estimating risks for not fulfilling them, falls under the domain of risk analysis.

9.3 Assumptions in the modelling framework

In our conceptual framework described in Chapter 5, we define a system correctness argument and a conceptual model. Here we extend them with the notion of modelling assumptions.

9.3.1 Assumptions in the correctness argument

So far, we assumed the system correctness argument as follows.

(*Model* **satisfies** P**) implies (***System* **satisfies** R**)**, where
P represents the set of model's properties, and R represents the set of system requirements.

In this chapter, we add assumptions to this argument:
((*Model* **satisfies** P**)** $\wedge$ A**) implies (***System* **satisfies** R**)**, where
A represents modelling assumptions.

Upon its discovery, an assumption can be either formalised to become a part of the formal proof, or can be non-formally described. In the latter case it supplements the correctness proof.

9.4 Classification of assumptions

We propose to classify modelling assumptions according to a number of criteria. The classes of assumptions serve as a checklist, or a template to fill while modelling.

The assumptions decomposition criteria comes from two areas. One comes from system engineering that uses system thinking to structure systems (Harel & Pnueli 1985)(Wieringa 1996) and identify different perspectives to be able to deal with the systems' complexity. The resulting classes of assumptions are the assumptions about the system components (C1) and system aspects (group of properties) (C2) . What exact components and aspect are, depends on the system domain. The designer can use the existing system decompositions or come up with their own. The advantage of the first one is that these structures are part of common mental model shared between the domain experts, and this decomposition does not have to be additionally introduced and explained.

Another class of assumptions coming from the system engineering and systems thinking are Constraints on the Plant and Environment (C3).

The other sets of decomposition criteria comes from logic and philosophy (epistemology), where not the system, but the knowledge is classified (Wieringa 1989). The classes of assumptions made are Necessary and Contingent Assumptions (C4, C5).

The reason why some assumptions are hidden comes from the lack of knowledge about the expertise domain of other designers. This is often seen in our own experience in practice.

The classes C1-C6 are further detailed in the list below. Classes C1, C2 answer the question *what* assumptions are describing. The class C3 focuses on the relevance for the system users. Classes C4, C5 focus on the criteria of their changeability. Note that these categories are not disjoint. The list of categories is not an ontology of the world of modelling, but a list of hints where to find hidden assumptions. Therefore, one assumption can be found in multiple categories.

C1: Assumptions about the system components. The requirement we want to verify determines where we draw the border between the system and its environment and what system aspects we will describe in the model. After that, we decompose the system, describe each component and, if necessary, decompose further. When decomposing the system, we simultaneously decompose the requirement, where each sub-requirement should be satisfied by a system component, and all sub-requirements together should imply the original requirement.

We can decompose the system in many different ways. We can make a process decomposition, a decomposition to the physical components, functional decomposition etc. The components can be described through assumption-requirement pairs in the form $assum(i) \implies req(i)$, where req is the sub-requirement we found while decomposing the system. For example: "If the wire is not longer than 12m (assumption), then the signal strength is sufficient for correct transmission (requirement)".

C2: Assumptions about system aspects. A system aspect is a group of system properties, usually related to one knowledge domain, one engineering discipline or one non-functional property. An embedded system has electrical, chemical etc. aspects, which are aspects defined by engineering disciplines. But we can also have a safety aspect that cross-cuts disciplines, functions and components. When designing the control and verifying the system requirement, we might need assumptions coming from these different knowledge domains. If, e.g., we are designing a shut-down system procedure for an embedded system, we want to know the electrical characteristics like capacity and resistance of the circuit that delays power-off, to calculate the time the procedure has to save the data.

C3: Constraints on the Plant and Embedded System Environment Some of the assumptions we make pose norms on the Plant and the users. We cannot be sure in advance that they will be fulfilled. The best we can do is to list them and deliver them together with the system. These assumptions are not part of the model - they can be seen as a label on the 'delivery box' of the system. For example: "If the weight in the cabin is larger than 20 and less than 150kg, the lift will go to the floor determined by the button pressed in the cabin."

C4: Necessary assumptions. These are the assumptions that always hold true.

Natural laws, like for example physics formulas, are considered to be true. If we have a system with a conveyor belt that transports bottles from the filling place, we will assume that its users will put the conveyor belt on a horizontal surface, respecting the law of gravitation and thus enabling them constant speed (the surface is levelled and the bottles do not roll down). Some of the plant components can be described with *engineering formulas* which we do not doubt. For example, the signal transmission through fibre optic cable is described with formulas that precisely calculate optical signal properties.

C5: Contingent assumptions. Depending on the context in which we use the system, some of the assumptions we take as true and do not consider them as changeable.

Contingent truths, unlike natural laws, on the other hand may change. There are some facts about the system for which we are not sure whether they will change or not. In practice, it often happens that the plant and the control software are designed concurrently (and are integrated in later stages). At this stage the plant design exists, but the detailed design may be determined later. For example, if we have a conveyor belt that has to move faster than originally planned, we can replace the existing motor with a more powerful one. (Then, we would have to change some parameters in control law implemented in control software. So apparently, these parameters were contingent on assumed plant properties.) Another example is that the plant is fixed, but our knowledge about it is changed. A domain expert can provide an improved formula describing the system behaviour.

C6: Assumptions made while co-modelling When designing a cyber-physical system, engineers of different backgrounds collaborate. In co-modelling

and co-simulation, a model is composed of two different models, for example one is software and one is control model. These two models together model or simulate the system. Some assumptions made while co-modelling may be coming from the different domain of framework of other disciplines. Or some systems properties are treated differently by different disciplines and if they are combined can end up in inconsistencies. We explore the example of software and control engineering creating one model (co-modelling). These two disciplines are an example of two disciplines using different modelling language, different mathematical frameworks. One designer may not even be aware she is making an assumption that is relevant for the other group. An example is the direction in which an angle goes from 0 to 360 degrees, one discipline could have it clock-wise and the other counter-clock wise. A similar example is the direction of coordinate systems represented in the Controller and mapped to physical system characteristics. They could go in different directions in two disciplines modelling the same physical positions or distances.

Criteria	**Classes**
Systems engineering	System physical components (C1)
Systems engineering	System functions (C1)
Systems engineering	System aspects (C2)
Systems engineering	Constraints on the Plant and environment (C3)
Logic and epistemology	Necessary assumptions (C4)
Logic and epistemology	Contingent assumptions (C5)
Logic and epistemology	Assumptions while co-modelling (C6)

Table 9.2: Criteria for finding classes of assumptions come from two different disciplines.

In the following section we list the (classes of) assumptions collected in the case studies.

9.5 Assumptions collected in case studies

In this section we give an overview of the assumptions found in the case studies described in previous chapters. They all have in common that the software was modelled to represent the plant and the environment.

For the first case we give two tables, as kind of two templates. One shows the list of assumptions and then maps them to the classes C1-C6. The other one starts with the classes and fills the assumptions.

9.5.1 Lego case

In the table shown in Figure 9.5, we list the assumptions collected in our first case study of modelling the Lego sorter.

Table 9.6 shows the classes of assumptions first and repeats the assumptions within each class. Note that an assumption can belong to more than one class.

After collecting these assumptions, the confidence in the model grows. As there are many possible assumptions about the system and the environment, the classification helps to come up with them and structures their collection.

Lego sorter: Assumptions and their corresponding classes
A1: Computer hardware works properly (does not break). **C1, C4** (infrastructure HW component)
A2: The operating system supports the control that we design. **C1, C4** (infrastructure operating system component)
A3: The sampling period is such that control can observe rotation angle with sufficient granularity. **C2** (Control aspect)
A4: The PLC controller supports the desired sampling frequency. **C1, C2** (PLC component, Control aspect)
A5: An operator will put the bricks in the Queue. **C3** (Environment)
A6: There are only blue and yellow bricks in the Queue. **C1** (Component: bricks)
A7: The motor moving the Belt is working properly (does not break). **C1** (Component: motor)
A8: The Scanner observes the colour continuously (not interrupt-driven). **C1, C2** (Component Scanner, Control aspect)
A8.1: The Scanner transmits the signal with the delay D1. **C1, C2** (Component Scanner, Control aspect)
A9: The rotation sensors and motors work properly (do not break or stop) and transmit signals with no delay. **A9.1:** The rotation sensors show only relative arm position. **C1** (Components: Motors, Sensors, Arms)
A10: All parts in the connection domain (motors and sensors) are working properly and introduce no delay **C1** (Components: Motors, Sensors)
A11: The sorter starts with predefined arms' initial position (they are not blocking the bricks entrance to the sorter). **C1, C3** (Component: Arms, Initial positioning)
(Continued on next page)

Lego sorter: Assumptions and their corresponding classes (Continued from previous page)
A12: There is a minimal distance between bricks so that **A12.1:** There is always "nothing_at_scanner" observed by Scanner before the new brick is observed and **A12.2:** There will be no new brick in front of the Scanner before a previous brick moves to the Sorter. **C1, C2** (Components and positioning)
A13: The order of a brick observed at Scanner and a new brick entirely on the Belt is not deterministic. **C1, C2** (Moving bricks, Aspect)
A14: Upon start, the Sorter is empty. **A14.1:** The system starts with no brick on the Belt. **C3** (Initial condition)
A15: The order of a brick arrival to the Scanner and previous brick sorted is not deterministic. **C2, C5** (Aspect and contingent truth)
A16: Bricks are standard Lego bricks (50mm x 15mm x 7mm) that fit in the Queue. **C3** (Constraint on the component)
N1: If on the Belt for T1 seconds, and the Belt is moving, the brick changes its position. **N2:** If the Belt is stopped, and the brick is on it, it won't change its position. **N3** The Belt speed v is a function of the value m put : on the Belt motor v = k x m **N4:** A brick does not change its colour. **N5:** Bricks cannot overtake each other. **N6** The plant has to be put on a flat surface **C4** (Engineering formula and natural laws)
Contingent truths: We might replace the Scanner that for example distinguishes more colours or that has a zero delay. The same holds for other plant assumptions; we might, for example, reconstruct the Plant in such a way that the Sorter arms have to move in different direction when sorting

Figure 9.5: Assumptions recorded while modelling Case study 1 (Lego sorter)

Classification C1: Assumptions about components		Classification C2: Assumptions about aspects		Classifications C3, C4: Necessary and contingent assumptions	
Components	*Assumptions*	*Aspects*	*Assumptions*		
Queue	*A6:* There are only blue and yellow bricks in the Queue.	**Control eng.**	*A3:* The sampling period is such that control can observe rotation angle with sufficient granularity. *A8.1:* The Scanner transmits the signal with the delay D1 *A9:* The rotation sensors ....transmit signals with no delay. *A9.1:* The rotation sensors show only relative arm position.	**Natural laws**	If on the Belt for T1 seconds, and the Belt is moving, the brick changes its position. If the Belt is stopped, and the brick is on it, it won't change its position.
Belt	*A7:* The motor moving the Belt is working properly. *A12:* There is a minimal distance between bricks so that *A12.1:* There is always "nothing_at_scanner" observed by Scanner before the new brick is observed and *A12.2:* There will be no new brick in front of the Scanner before a previous brick moves to the Sorter. *A13:* The order of a brick observed at Scanner and a new brick entirely on the Belt is not deterministic. *A15:* The order of a brick arrival to the Scanner and previous brick sorted Is not deterministic.	**Mechanical**	*A1:* Computer hardware works properly. *A12:* There is a minimal distance between bricks so that *A12.1:* There is always "nothing_at_scanner" observed by Scanner before the new brick is observed and *A12.2:* There will be no new brick in front of the Scanner before a previous brick moves to the Sorter. *A13:* The order of a brick observed at the Scanner and a new brick lying entirely on the Belt is not deterministic. *A15:* The order of a brick arrival to the Scanner and previous brick sorted is not deterministic. *A16:* Bricks are standard Lego bricks (50mm x 15mm x 7mm) that fit in the Queue.	**Engineering formulas**	The Belt speed *v* is a function of the value *m* put on the Belt motor: v = k x m
Scanner	*A8:* The Scanner observes the color all the time (not interrupt-driven). *A8.1:* The Scanner transmits the signal with the delay D1 *A12, A13, A15*			**Contingent truths**	Here, all the descriptions about the Scanner are contingent. We might replace the Scanner that for, example distinguishes more colours or that has a zero delay The same holds for other plant assumptions; we might, for example, reconstruct the Plant in such a way that the Sorter arms have to move in different direction when sorting
Sorter	*A9:* The rotation sensors and motors work properly and transmit signals with no delay.. *A9.1:* The rotation sensors show only relative arm position. *A11:* The sorter starts with proper initial position. *A12, A15* *A14:* Upon start, Sorter and Belt are empty.	**Electrical**	*A7:* The motor moving the Belt work properly. *A9:* The rotation sensors and motors work properly *A11:*The sorter should start with proper initial position.	**Classification C5: Plant and environment constraints**	
Bricks	*A16:* Bricks are standard Lego bricks (50mm x 15mm x 7mm) that fit in the Queue. *A12, A13, A15*			*A5:* An operator will put the bricks in the Queue. *A6:* There are only blue and yellow bricks in the Queue. (If a brick of any other colour is there, the Scanner will recognize it as Yellow or blue and will sort it on one of the sides.) *A11:* The sorter should start with proper initial position.	
Operator	*A5:* An operator will put the bricks in the Queue. *A11:* The sorter starts with proper initial position. *A14:* Upon start, Sorter and Belt are empty.	**Implementation**	*A2:* The operating system supports the control that we design. *A4:* The PLC controller supports the desired sampling frequency. *A8:* The Scanner observes the color all the time (not interrupt-driven).		
Controller hardware and OS	*A1:* Computer hardware works properly. *A2:* The OS supports the control we design. *A3:* The sampling period is such that control can observe rotation angle with sufficient granularity. *A4:* The PLC controller supports the desired sampling frequency.				

Figure 9.6: List of the assumptions shown according to the classification criteria we found.

9.5.2 Paper inserter case

The list of assumptions in the paper inserter case is shown in the table in Figure 9.7. In the column "Category" the assumption category is indicated.

Here as well, the classification helped to collect relevant assumptions. The practitioners that we interviewed found them very useful. One of the interviewees made a remark that the more obvious an assumption looks to one domain expert, the more it is difficult when the condition is not fulfilled.

Paper inserter case: Assumption	**Category**
Rollers always move in one direction.	Plant Mechanical eng. domain
The shape of the path is not relevant (It can be straight or with curves).	Plant part Mechanical eng. domain
The material moves parallel to the belt length. It does not move to the left or right.	Plant part Mechanical eng. domain
The vertical or horizontal position does not play a role. Gravitation does not play a role.	Plant process Mechanical eng. domain
If the clutch is off, right after the trigger out, the track component will continue to move the paper.	Plant part System eng. domain
The slip is constant on one track for all the rollers and all the papers.	Plant part Mechanical eng. domain
A roller has two wheels. (in the model represented with one slip).	Plant part Mechanical eng. domain
A track can be with or without a slip.	Plant part Mechanical eng. domain
Turning the sensor on and off is instantaneous.	Plant part Mechanical eng. domain .
The length of the sensor equals one pulse.	Plant part System eng. domain
Mechanical characteristics of the feeder determine the separation distance which stays the same all the way through the feeder module	Plant part Mechanical eng. domain
There is always paper in the feeder.	Plant part Environment constraint
Track slip does not change for a track	Plant part Physics law
There will not be two triggers at the same time. If this would happen, the model will recognise only. the first one.	Software control constraint
(Continued on next page)	

Paper inserter case: Assumption (Continued from previous page)	**Category**
No plant component can select between two pieces of material. The model calculates which data to forward.	Software control constraint
Switch On and Off of the Distributor are the only two of its positions. Changing from one to another happens instantaneously.	Plant part Mechanical eng. domain
Three papers cannot be combined (e.g. paper from feeder 1 and two papers from feeder 2)	Plant product constraint

Figure 9.7: Assumptions recorded while modelling Case study 2 (paper inserter)

9.5.3 Printer case

The list of assumptions in the Printer case is shown in table in Figure 9.8. In the column "Category" the assumption category is indicated.

Printer case: Assumption	**Category**
There will not be a paper jam.	Plant constraint
Power is standard.	Plant environment constraint
Off and Sleep state are the same.	Software control sub-systems
Delays outside production state are allowed. are allowed.	User requirement assumption
There is no network delay.	Plant environment constraint
There will be no overload of print requests.	Environment (operator behaviour)
The Cleaner temperature is constant	Plant component
Paper transport time is much smaller than printing time	Environment Plant
All printing tasks take the same time	Plant

Figure 9.8: Assumptions recorded while modelling Case study 3 (Oce)

The list of assumptions should be checked with the domain experts (the domains are indicated in the columns of tables) to check if they are correct.

9.6 Destecs case - co-modelling

In this section we take a closer look at the classes of assumptions made while co-modelling. They may result in incompatibilities, inconsistencies and incom-

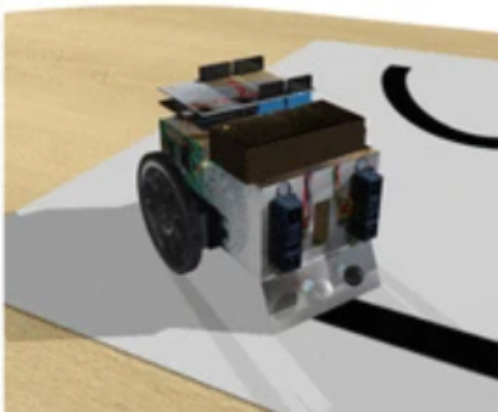


Figure 9.9: Line-following robot.

petencies while co-modelling control and software. An inconsistency here is an inconsistency between two assumptions made by different groups about the same system. We take the example of software and control engineers co-creating models. The example that we will use is the system developed in the research project Destecs (Broenink 2010)(John Fitzgerald 2016). The project developed co-modelling methods and tools, and when we joined the models already existed. We studied the models, talked to its designers and collected assumptions that were made while modelling.

The project's pilot study of modelling a line-following robot. The robot is a small cart with two wheels, each moved by a servo motor. It is shown in Figure 9.9. Also, each wheel is connected to an encoder that senses the wheel's angular position. At the robot's front there are two infrared sensors that sense the lightness on the floor based on infrared reflection. Also, two infrared distance sensors mounted on the robot sense objects based on reflected infrared light. A contact switch detects bumping to another object or a wall. The requirement for the robot is to follow a black line drawn on the ground, using two infrared sensors, and stopping in case there is an obstacle in front of it.

As it is the case in a pilot study, different models were designed, with different software and controller architectures, and while this was done, the control and software experts learnt from each other about the other domain.

The high-level model structure is shown on Fig. 9.10. This is a high-level diagram, with many details abstracted away as they are not relevant for our discussion. For example, in the CT (control model modelled by control engineers using linear control theory), the loop control does not actually show the feedback loop, and the co-model consisting of the control and software model does not directly connect the contract with the two models, but they are equipped with interfaces towards each other (these are two simulation models "talking" to each other via shared variables on their interfaces). Also, the DE (discrete

software-implemented) control architecture is not given in our diagram, but will be discussed later.

The plant is modelled using bond graphs, typically used by control and mechatronic engineers to graphically represent a physical dynamic system. Such a graph typically transforms electrical, control and mechanical variables into domain independent concepts connected in a graph. A bond graph describes a physical system as a number of physical concepts (the elements) connected by energy flows (the bonds). An example is shown in Figure 9.11. This example, copied from (20-sim 2023) shows how bond-graph is derived. A car suspension system is modelled using springs, masses and dampers. In this example masses M1,2, spring (k) and damper (D) characteristics are transformed into a graph describing flow of energy (the energy can be electric, mechanic, hydraulic etc but the graph looks the same).

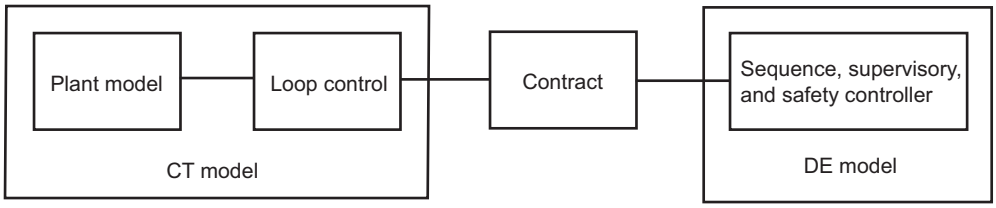


Figure 9.10: High level diagram of a co-model.

The environment of the robot is the ground with a black line is described with the file that contains the lines coordinates. The loop control is distributed to two processors, each controlling one motor. It is a PID controller, well known in the theory and practice of control engineering. Its name stands for Proportional-Integral-Derivative that describes kind of correction in the feedback loop. It usually asks for tuning its parameters in simulation and by doing experiments. We will not go into the details of it, as it is not relevant for the case.

The control steers the motor of the wheels via implementation on the DE side. Also, the part of the control that deals with different faults is on the DE side.

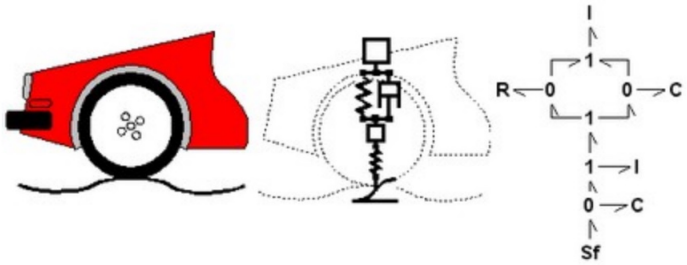


Figure 9.11: Example of a bond graph

The contract describing the communication on the interface between CE and DT model contains parameters and variables that are read by one of the models and written into by the other. The parameters shared in this particular case are the lateral and longitudinal offset of the robot. The variables that are shared are the values of encoders, which the CT side writes into, and the DE side reads. Another set of values read by the DE side, are the values of infrared sensors. The DE side writes into the variables that set the referent speeds for the right and the left wheel (in case of a curvy path, the wheels will have different speeds).

The contract also gets the information about the faults detected from CT side, and the value of LED diodes is written by DE side, so that the operator gets the signal if there is a fault and when the robot is in regular working mode.

9.6.1 Classes of incompetence and inconsistency

Below we give a list of sources of inconsistency and incompetence. We cannot assume that we cover all of them, but a model is only consistent (competent) if it avoids all possibilities of inconsistency (incompetence).

Ideally, we would like to have all sources of inconsistencies and incompetences to be automatically detected by a tool. Unfortunately, this is not possible in general. Some can be found by a tool straightforwardly (syntax check), for some we could add information to make them automatically detectable, as, e.g., by extended types. Still a major part cannot be automatically treated. For these we suggest to use the list below as guidelines, in the sense that if we address issues listed there explicitly, we decrease the chance of getting inconsistent and incompetent models.

Incompetence and inconsistencies can be introduced in initial models or during the evolution of models. As this does not make a principle difference for the sort of incompetence and inconsistencies, we do not distinguish these two phases in our analysis.

Contract level syntactic inconsistency:
A co-model is syntactically inconsistent if both models do not address the same set of variables, parameters and events, with identifier names, types and ranges.

Contract level semantic inconsistency:
If the variables, parameters and events do not address the same phenomena in both models, the co-model will be inconsistent, e.g. a variable that represents speed of a wheel is interpreted differently in the sub-models. For example, a control model can use it to transform it into a distance using a different equation than the software model does, which results in a different value for the distance. In the motion systems and autonomous vehicles, it is very important to always interpret correctly spatial information. Studying the documentation of the robot case, we found that the two groups of experts interpret differently the angle values, the CT modeler assigned the positive angle in the anti-clock wise direction, whereas the DE modeller assumed that the angle goes up in the clockwise direction. Therefore, for these kinds of systems it should be established

beforehand a referent system and the initial robot position and orientation in this referent system.

In the robot two encoders are connected to the right and the left wheel, and it is important that their connected with the correct connectors of the interface card, which is a piece of hardware used to connect the encoder and the CPU. Otherwise their identities can be mixed (the left wheel can be read as the right while and vice versa). In the robot case, there are only two wheels to control, but in the systems where many units of the same kind (e.g. beds in hospital connected to a central computer that monitors patients) can be easily accidentally wired wrongly.

Also if the measurement units do not coincide, models are inconsistent.

Variable propagation inconsistency:
It might be the case that at the interface/contract variables are interpreted consistently, but , e.g., while passing them to other components of the control software, the semantic interpretation of the variable changes, leading to inconsistency.

In the robot model, we noticed that the DE model reads the variables values and then distributes these values from one variable to another. In copying the value from variable 'a' to variable 'b', and then copying the value of the variable 'b' to variable 'c' and so on, it is extremely important to keep in mind what this value represents. Theoretically, it could happen that the encoder value is copied into the variable that represents the speed or the variable that represents the value of a different encoder. In case of robot, if it would follow a straight line, this would not be easily noticed, as the two wheels will always have the same speed.

Initialisation phase inconsistency:
For the initial phase of a simulation run (and also a real system execution) assumptions are made on the initial values of variables, about start-up procedures (e.g. a control may assume that a robot starts up with the robot head in zero-position. In reality the position of the robot-head is where it was at the end of the last robot operation. Finding out the real position of the head may be part of a start-up procedure that is called homing), and the availability of values in buffers.

Mode change incompetence:
A plant may operate in different modes, e.g. line following mode and turning mode for a robot. When changing modes we also have assumptions on variable values for the new mode starting. In a generalised view, this may also be considered as an initialisation phase inconsistency, but we want to distinguish it here for explicitness.

Usually there is a start up mode, during which the plant moves to stationary, working state; then there is the working mode that also can consist of different modes; finally, shut-down mode puts the vehicle into a safe position.

Control theory assumes linear systems within of a mode, but the switch from one mode to another can potentially be non-linear, which may bring unwanted behaviour. The physical behaviour of a plant is possibly not in a steady state immediately after mode change, but needs a time to achieve a steady state. In

control engineering we speak "bumpless transfer" when these transition phases are properly addressed. An example is the robot in the line following mode, that will need some time to reach the intended speed. This is a common phenomenon occurring in practical control design. If it is not addressed, it will lead to incompetent models. If only addressed in the plant model, and not in the control model, we will have inconsistent sub-models.

An example of the transfer between modes may mean unwanted sudden changes of currents and voltages, but it can also have consequences in not controlling the plant parts. For example, a vehicle similar to our robot was designed to carry packages along a partly inclined surface. Two different control modes were used to control the vehicle, and as the vehicle climb with a high speed, the packages bounced off it, when it reached the horizontal surface. The bouncing was an uncontrolled phenomena, left there as it did not pose any danger. But, these 'in-between' modes behaviours need to be carefully examined, and sometimes control has to change to avoid unwanted or unpredictable behaviour.

Summary of assumptions made while co-modelling
Contract level syntactic inconsistency
Contract level semantic inconsistency
Variable propagation inconsistency
Initialisation phase inconsistency
Mode change incompetence
Sampling time incompetence
Computation time incompetence
Quantization incompetence
Boundary incompetence
System/environment inconsistency
Abstraction inconsistency
Idealisation inconsistency
Idealisation incompetence
Modelling trick inconsistency

Figure 9.12: Assumptions recorded in Destecs case

Sampling time incompetence:
Control theory states that for the discrete control of a continuous plant a certain sampling period is necessary to ensure that the plant behaves as intended. The sampling time depends in the first place on the dynamical plant behaviour and the physical properties of the plant. As there is an infinite set of possible dynamic plant behaviours, the requirements determine the ones of interest. If the correct sampling time is not considered in the control model, the overall model is not competent.

Computation time incompetence: Limitations of the control platform (hardware and operating system) may lead to jitter, i.e., variation of delays in reading sensor values and writing actuator values. This may lead to violation of the hard

real time constraints of the control law of the computation, i.e., the next steering value is not available before the next sampling moment. There is a number of solutions to the problem. If the problem is not addressed, the submodels may not be competent. If the chosen solution is not communicated from the control engineer to the software designer, then there may occur inconsistencies within the DE submodel.

The robot control law takes into account the calculation time, so the calculation based on the measurements in the moment k*T, is calculated to be written into actuators in the (k+1)*T moment in time, which is the next sampling time.

Quantization incompetence:
A sensor measuring some quantity gives typically an analog value as result (voltage). By quantization, i.e., digitalization of an analog value, the accuracy of the value decreases (i.e. the accuracy depends on the (size of the) representation). If the DE model does not cope with this inaccuracy, the DE model will not be competent. An example may be that we integrate about (inaccurate) speed to derive the position of an object, which will by "wrong" to the extent of the inaccuracy of the speed.

Boundary incompetence:
When a quantity to measure is beyond the maximum level of the sensor, then there are different ways how the sensor writes its measurement. One possibility is that the sensor keeps writing its maximal value. If this maximal value is, e.g., misinterpreted as a constant quantity, then this may lead to an incompetent or inconsistent control, i.e. an incompetent or inconsistent DE model.

System/environment inconsistency:
Often, we do not model a "complete" system, but only a fragment of it (e.g. when modeling a car, a nuclear power plant, etc.). It has to be well defined what the borders of the system that we model are, and what assumptions we have about the environment (e.g. when there are shared resources, about the availability of these resources, e.g. for the robot: what is friction of the ground, is the ground flat, is there an inclination? e.g. for a automotive system: is the ECU used available often enough, do the buses used have enough capacity?) Both sub-models have, of course, to address the identical fragment of the system. This may be trivial for toy examples, but for real cases it is non-trivial.

Abstraction inconsistency:
Abstraction is one of the basic techniques of modelling, "leaving away" non-relevant aspects. It has as a consequence that all models are "wrong", but what we try to achieve is that we still can deduce some properties of the real system from the properties of the model. Obviously, if we make different abstractions in the sub-models, the co-model may become inconsistent. As an example we take here the case where friction and torque are modelled in the CT model of the robot, where it is possible that under certain circumstances slip occurs (wheels rotating without the robot moving). A control model on the DE side that has abstracted away from slip, may interpret the movement of the wheels wrongly and come to wrong conclusions about speed and position of the robot.

Idealisation inconsistency:
Idealization is similar to abstraction. But, whereas in abstraction we leave some-

thing away, for an idealisation we introduce simplified behaviour. The classical example is the point mass modelling in physics. In reality, there are no point masses, but for modelling it is a useful idealisation, that allows for a simpler mathematical description, and gives satisfactory results. A typical idealisation in control is that there is no delay in reading, writing or computing. If idealisations on delay are different in submodels, this may lead to inconsistency.

An idealisation typical for control design is idealising the plant as linear. The plant model is derived from the first principles (established laws of physics), or empirically (in black box modelling process) and it is not entirely accurate. There are two reasons for having an inaccurate plant model. First, it is sometimes impossible to derive an accurate plant model. Second, even when this is possible, such a model would be very complex, too complex to manipulate it and to derive the control. So, a simplified model is derived, simple enough to manipulate it, but complex enough to give predictions that are accurate enough.

This does not only enormously simplifies the control design, but it also makes it a lot easier to reason about the system. It would be very hard to understand the dynamics of a system without

first focusing to the special case of linear systems. When modelling a system composed of electrical, hydraulical, mechanical, optical and so on components that interact with each other, the modeller starts with linear systems, and then, if necessary, extends the model with non-linear components.

If we would extend our robot to be a part of a network of autonomous robots that can communicate with each other, there would be a potential danger of electro-magnetic interference of the vehicles' sensors. The plant may have to be re-modelled, which as a consequence would need a different control laws, which would again affect the higher control layers modelled on DE side.

Idealisation incompetence:
A typical example from control engineering is a robot that bumps into a wall. Physically, this is a phenomenon with very fast, but continuous dynamics. The robot's speed and acceleration during bumping change with a very high frequency. If the details of this transition from one steady state (before bumping) to the next steady state (after bumping) are not of interest, bumping into the wall can be represented in the model in a much simpler way. For example, in hybrid systems community, it is common to model it as an event that changes the speed and acceleration values from some value to zero, instantly.

Modelling trick inconsistency:
Tools provide languages to model a system. Typically, these languages are limited in their expressiveness, and often we have to squeeze a model into the modelling language of a tool. Experienced users of a tool know "the way how to model for this tool". A modelling trick describes a different behaviour that the "natural" one, simply because of the expressiveness limitation of the tool. If the modelling trick is not made explicit to the other domain modeller, an inconsistence co-model may be the consequence.

In Uppaal, for example, we add additional variables, states or automata just to be able to formulate logical expressions that check properties of interest. If we would use the model in combination with another model, it should be made

clear that these additional elements do not represent elements of the system, even if they look like they do.

9.7 Usefulness of collecting assumptions

To conclude this chapter, assumptions, when stay implicit may result in an incorrect model. Implicit assumptions are typically not taken into account by all the designers and the domain experts, and even if they are they might slightly differ. We therefore propose to collect the most important assumptions in a list to guide collecting them. This list can be used for checking among the designers if they agree on their truthfulness.

Another benefit of collecting them is that in the future, when the new variants of the system are being designed, it may be worthwhile checking if they still hold. If not, then the model may have to change.

The classes we propose are meta-classes, and it is up to the modeller to instantiate them to the classes that apply to the modelled domain. For example, the class C1 - components, can refer to physical components, to processes, functions. This is usually done to break down the complexity and it will probably correspond with one of the architectural views or decompositions.

Do we have to write the necessary truths? Not always. It is not likely that we will write the law of gravitation. Some of the necessary or contingent truth are created by the domain specialist that experiment with the system and describe the behaviours for which there are not exact formulas. For example, in some of the machines applying chemicals to hard surfaces, there is no one formula that describes the process. By building the prototype and measuring desired parameters, engineers derive some analogues functions within the range of temperatures, humidities, and pressure of the environment where it will be used.

Finally, when the model consists of two models made by different experts using different frameworks, they are designed to be interoperable. We gave the example of the models designed by control and software engineers. But in practice, there are many more examples.

9.8 Summary of this chapter

The result of this chapter is extension of the design problem to design a justification argument with the technique and guidelines to capture the modelling assumptions. At the same time, the steps proposed in the guidelines answer the conceptual question on what constitutes a justification argument.

Figure 9.13 sketches the scope of this Chapter.

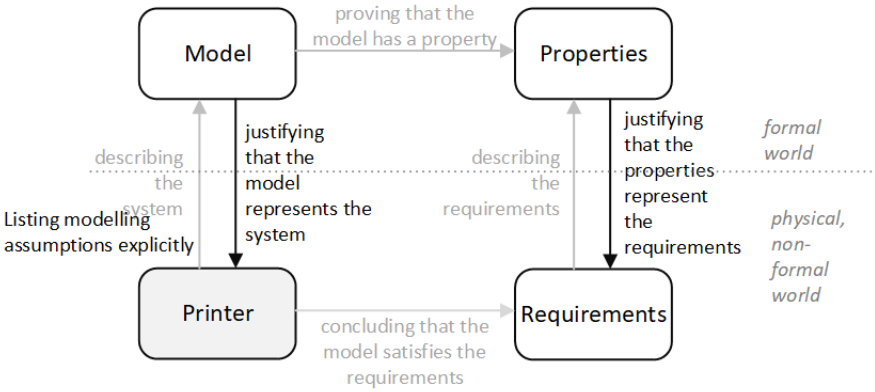


Figure 9.13: Modelling assumptions as part of the justification argument

In this chapter we provide the following refinement of dealing with the assumptions.

- Classification of assumptions we collected in the cases (generalisation) to organise assumptions gathering
- Checklists for gathering assumptions
- Updated structure of the justification argument

Chapter 10

Modelling method

In this chapter we consolidate all the findings and the updates of the initial version of the method presented in Chapter 5. We end the chapter with answering the final question presented in Chapter 1 (Table 1.1), and that is the following knowledge question: *To what type of cyber-physical systems is this method applicable?* The summary provided in this chapter can be used as a high-level guideline/reminder when using the method in practice.

10.1 When to use this method

In Chapter 2, we explained in detail the scope of the method and the motivation for it. We summarise it here before giving the overview of the method.

The scope. The method aims to improve the verification of the software controller specification by improving the way its model is designed. The method can complement existing formal and semi-formal verification modelling techniques, methods and tools.

Kinds of systems the method applies to. The modelled systems are mechatronic systems, consisting of electro-mechanical plant controlled by software. There may be other aspects of the plant made by other engineering disciplines, such as chemical engineering parts. These others disciplines have not been explicitly addressed in our case studies. The systems for which we designed the method typically move and/or transform workpieces (products). The system requirements that are modelled are behavioural, functional requirements, modelled with state machines or automata.

Why this method? Modelling a software Controller includes modelling the Plant and the system's environment (Jackson 2000). This is because we are building the Controller that will together with the Plant fulfil the system Requirement. The system Requirement describes both the Plant and how the Plant reacts to its Environment. The system's relevant environment and the Plant are also called Problem Domain. (The Plant and the system relevant environment are together the Controller's relevant environment. To avoid con-

fusion between the system's and Controller's relevant environment we call the latter a Problem Domain.)

Many software modelling methodologies provide languages, tools and techniques for model design. Typically, the focus is on the formal part and on the final outcome (the model). Often, the process of modelling that results in a good quality model is left implicit and it is considered something learned, something that comes as a result of experience and talent. Whereas this is certainly true, that a good modeller needs to learn by doing, this thesis proposes a structured method to learn modelling and to structure the model design process. Whereas the creative process cannot be entirely put into a framework and process, some of its parts and best practices can. Modelling is more than an art.

Even though the majority of the work in model-based software engineering is in formal aspects, there are researchers that cover some parts of the non-formal ones. In Chapter 4, we give an overview to related work from the software and system engineering. Additionally, we reach to philosophy of science that has been studying models.

Hall & Rapanotti (2016) designed a theory for software engineering in which they focus on the (software) design process and formally describe the steps of combing to a software solution through transformation on stakeholder need / problem into the solution. (Hall & Rapanotti 2008a) also formalise the justification of the designed solution, which helps specially in safety-critical context. In their work, the authors acknowledge that there is a step in which the stakeholders is convinced enough (which is informal step), but justification of the solution is with these steps documented and increases the confidence in the solution. Our work complements to theirs with taxonomy of modelling decisions and a method to classify assumptions.

In the system engineering community, (Sandee et al. 2006) propose threads of reasoning when creating informal models used in system engineering, supported by different techniques for deepening insight (story telling, 5-why technique, among others). (Haveman 2015) provides a framework for guiding decision making and communication of the modeller and domain experts.

Researchers of conceptual modelling, such as (Robinson 2008) and (Thalheim 2012) name the steps when designing models and validating them.

As explained in details in Chapter 4, philosophers of science have been looking at relations between the model in science and the real world phenomena. (Boon 2020) proposes a pragmatic view, a step from representativeness view, to a method described in (Boon & Knuuttila 2009) and extended in (Boon 2020).

Our contribution is in naming the steps when designing the models designed for the purpose of system verification, as well as providing a method to structurally capture modelling assumptions.

What modelling activity does the method apply to? When designing the software control of the system, formal methods can be used. The control of the system (model) will always "mimic" some part of the plant and the environment. Good modellers make these models not more complex than necessary. Some part of this craft and good practice we turn into structured steps, a framework to follow and argumentation structure to increase the confidence in the

model. Therefore, this method can be used to guide the model design, to explain the model or to teach others how to make good models. In the case studies we performed, we used the steps to create a modelling handbook (Chapter 7) for reusing modelling decisions, in another we used it to communicate the model with the domain expert (Chapter 8), and in the third case we identify steps to increase the confidence in the model (Chapter 9).

How to use the method? The method can be used as a checklist, or a modelling process template. It is meant to accompany the existing modelling languages, tools and techniques. It can be used while designing the model and when explaining the model to another expert. In the case studies we performed, the modelling handbook we designed had the format of questions/checklist (Chapter 6), together with reusable model elements. In other cases we used the steps to explain the model and in cases described in Chapter 8 and 9, we used the tables with the modelling assumptions categories to check our assumptions.

Who should be using this method? First, the modeller to create the model and to validate it by explaining it to the domain experts. It can also be used by the users of the model to explain the model. The model is used for verification purposes as well as communication.

10.1.1 Positioning the method in requirements verification - going down the V-model

The end-goal of verification is to show that the **customer (user) goals** can be satisfied by using the system. The **top-level functional system requirements** describe what the system does, in terms of the phenomena of the Plant and the environment. The requirements of complex systems are typically decomposed into **sub-requirements**. Our method is applied for a chosen requirement in this goal/requirement tree that describes the system behaviour. Furthermore, as described in Chapter 2, we assume that the Plant already exists, and that its Controller is being verified.

10.2 Method overview

Figure 10.1 provides an overview to the method we designed. This picture shows three main ingredients of the method. From left to right, they are as follows.

1. A **process** to follow to design an adequate model and verify the requirements. There are three phases: modelling problem analysis (or post problem analysis), model design and model justification. Below we open each of the boxes depicted in this picture and explain it in more detail.

2. Top-level **reasoning structures** – reference models – that show main entities and their relations. We call them “reference models”, as they are generic, high-level conceptual models that can be used in combination with any formalism or tool.

3. Two other structures: the **taxonomy** of the model-system relation (and construction steps) and different **classifications** of modelling assumptions. These two are used in model design and model justification phases, respectively.

We will first explain the second component mentioned above: top-level reference (conceptual) models shown in the middle column of Figure 10.1. We will then look in detail into the modelling process, and accompanying taxonomy and classifications (criteria).

10.2.1 Top-level reasoning structures

Conceptual model with relations between modelling process entities

In Figure 10.2 four concepts and their relations are depicted. On the bottom half, there is a system for which we are verifying that the requirements hold. The system and the requirements represent in this diagram something tangible and observable in the "real", physical world. In Chapters 1 we described a luggage handler, with the requirement to bring the luggage to the right carousel at the airport. Another example: a printer described in Chapter 8 with the requirements that it prints papers within a specified time-range for the given scenario of print requests.

As explained in Chapter 2, the system itself consists of the Plant and the Controller; the Plant design is known and assumed to be verified and correct. We are in the role of software engineers, verifying that the Controller, together with the Plant, satisfies given requirements.

On the top half of this diagram, there is a model representing the system, with respect to the (functional) requirements; there are properties of the model representing the requirements. The model and the properties belong to the formal, mathematical world. The model is adequate if it represents the system correctly, with the respect to the purpose it has been designed for; meaning that, for the purpose at hand, if the conclusions we draw about the system by looking at the model, are correct.

As described in Chapter 9 there is always a number of modelling assumptions that are not described with the model. We define them as postulates about the environment of the control software being modelled, that are relevant for the model validity.

If the model represents the system adequately; and if the properties for the model hold (they are validated in the formal world); and if the assumptions in the physical, non-formal world are true; then we argue with high degree of confidence that the requirements for the system hold. The goal of modelling is not only to design the model but to build this logical argument.

Conceptual model with main modelling process phases

The conceptual model in Figure 10.3 shows some of the high-level modelling steps from Phase 2 (model design) and Phase 3 (model justification) from the

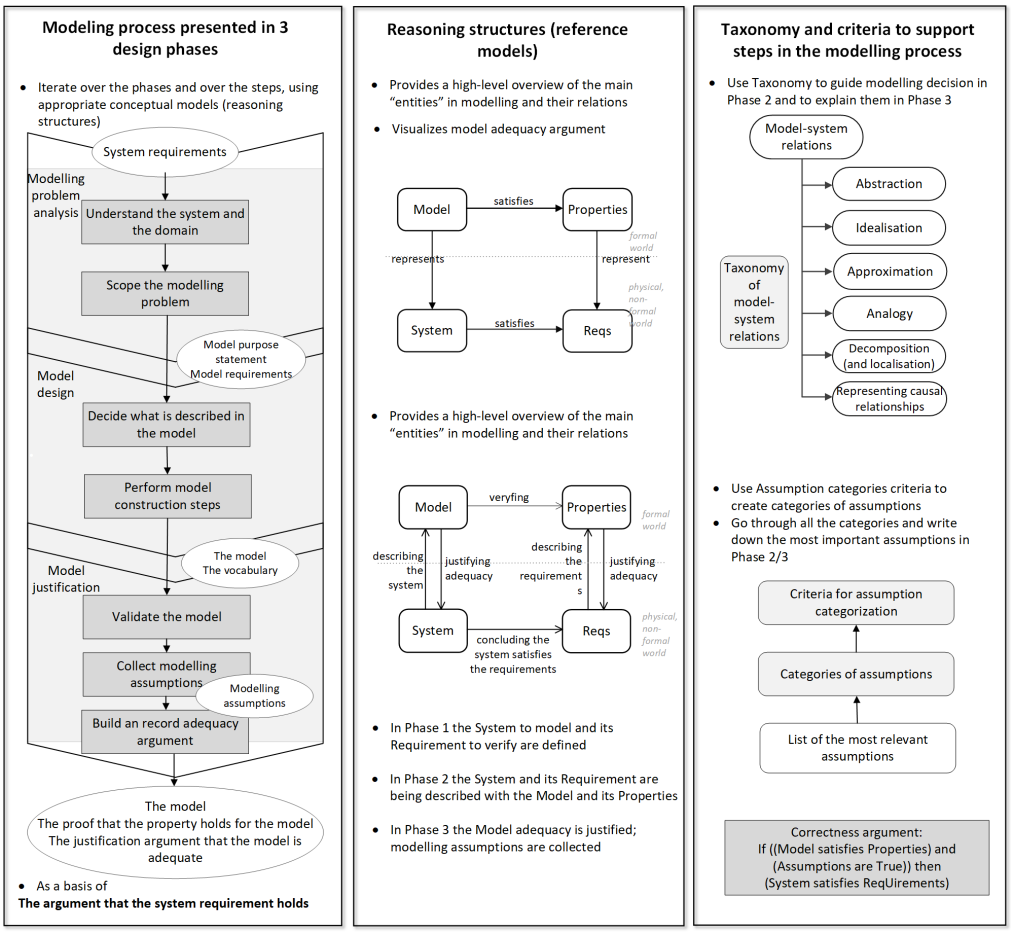


Figure 10.1: Method overview: The reasoning structures in the middle show the main concepts and their relations between them. On the left side, the process is shown, with modelling steps. Even though they are presented as sequential, the modeller goes through them in a non-sequential way. On the right side, guiding structures, taxonomies, and criteria are shown. We argue that the system satisfies the requirements in a logical argument structured, as shown at the bottom of the third column.

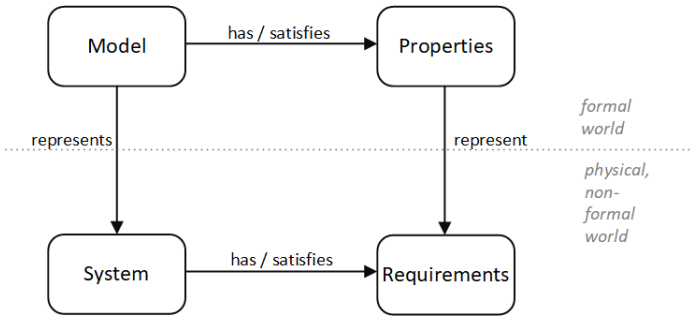


Figure 10.2: The conceptual model showing relations between the model, the system, the model's property and the system requirement.

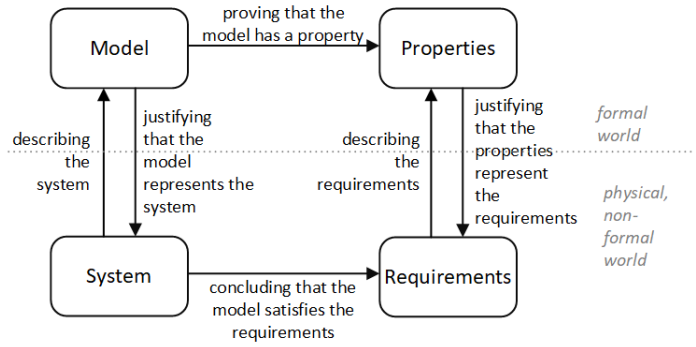


Figure 10.3: The conceptual model showing model construction and model justification steps.

modelling process. Describing the system and its requirements with the model and its properties are non-formal steps, part of Phase 2 in the modelling process. The steps of justifying that the model and properties are adequate are a mixture of some formal model validation steps followed by non-formal arguments and steps. Note that these are iterative steps. They are, as (Hall & Rapanotti 2008b) describe iterative steps between understanding the problem and designing the solution.

Model property verification is a formal step, and the conclusion that the system satisfies the requirements is part of a non-formal, structured argument we mentioned.

These two conceptual models outline and visualise what we are designing, what we start with, and how the modelling process cannot be purely formal.

10.2.2 Modelling process

We present the modelling process in three design phases. They are shown as sequential, but in practice, the modeller goes through them in an iterative manner, through phases and through individual steps, until they come to the final model and the argument that the system fulfils the requirements. The phases are as follows.

- Phase 1: Problem analysis - starts with (usually vague) description of the requirements and system to verify. In this phase, the goal is to define the following.
  - The purpose of the model.
  - What exactly is the system segment to verify and what is the requirement to verify.
  - What are the additional requirements for the model, and modelling (these are meta-level requests and constraints)
- Phase 2: Model design starts with the output of the previous phase and ends with the model, as well as the vocabulary with definitions of concepts.
- Phase 3: Model justification. In this phase the model is justified and modelling assumptions are consolidated and documented.

As it is usually the case when trying to represent the design process as a workflow, there can be, and there usually are many iterations, going back and forth between the phases and steps within them.

Modelling process - Phase 1

This phase starts with the system requirements that need to be verified. Often they are not precise enough, or need further refinement. As already stated, we are here in the role of the software engineer designing the Controller. The Plant is considered to be already designed, and the knowledge about it can be obtained. The knowledge is gathered from the domain experts, from the documentation. There can be tacit knowledge discovered while modelling or discussing the initial questions with the domain experts. We assume that complete knowledge about the Plant can be obtained. If not, we recommend documenting the assumptions made by the modeller or domain experts.

The two main outputs of this phase are the model purpose and the model requirements. The first one, the **model purpose**, states the definition of the system (segment) to model and of what requirements need to be verified with the model. The output of this phase is represented in the the bottom half of Figure 10.2. If the requirements are multiple and/or coming from multiple stakeholders, the modeller should decide whether to use one model for multiple requirements (and purposes) or multiple models. This is typically a pragmatic

decision, where often a trade-off between maintaining and evolving multiple models and a model simplicity and understandability are made.

Besides the segment to model and a (sub-) requirement to verify, there are other questions about the model purpose to answer: is the Controller being designed or only verified; is the model built for simulation or some other purpose; are we building the model to generate the code from it? Some of the answers to these questions come from determining who the stakeholders of the models are, who needs the modelling result, who needs to maintain it, and who needs to understand it.

The second output of this process is the **requirements for the model itself**. They include quality requirements for the model and their relative priorities. Sometimes in practice, there are practical constraints in what tool, formalism or language to use.

If the request is to verify certain requirements with a certain technique, the modeller as the expert will first decide whether this is possible. This requires the modeller to understand the domain, the system and controller architecture. Also, the modeller at this stage will have to determine whether the initial requirement is realistic, whether it needs to be decomposed, or refined into another requirement. There may be a negotiation with the model stakeholders to change the requirement.

One recognised thing that happens in practice is that the purpose shifts. The model can be reused for a similar purpose, or can be slightly extended to be able to verify another requirement. We therefore highly recommend to revisit the question about the model purpose often, and if it changes, to document it in the model purpose statement. Also, if it happens that the small change in purpose of the model has a big effect on the model, it may be better to create (and maintain and manage) a new model. We recommend to explicitly discuss the trade-offs when the purpose changes, for extending the model or creating a new one. This prevents unfulfilled expectations and unexpected surprises.

The model purpose - the system and requirements to model decision already decides on many abstractions to make. In Chapter 2, we step-by-step reduced the meaning of the Controller to the high-level process controller. We also mention the work of Jackson to distinguish between the Plant (Domain) and the Requirement. These structures, together with the precise definitions in WRSPM framework, can be used to force us to be precise when defining the System to model and the Requirements.

Important in this approach is the recognition that, the cyber-physical systems software supports achieving the system requirements that are visible in the Plant and in the environment. As such, the software model internally represents the Plant, "mimics" it, to be able to control it. We capture this in extended problem frames diagram shown in Fig. 10.5. We extend this model to represent the fact that it is the plant representation within the controller, and on the modelling level, as a model within the model.

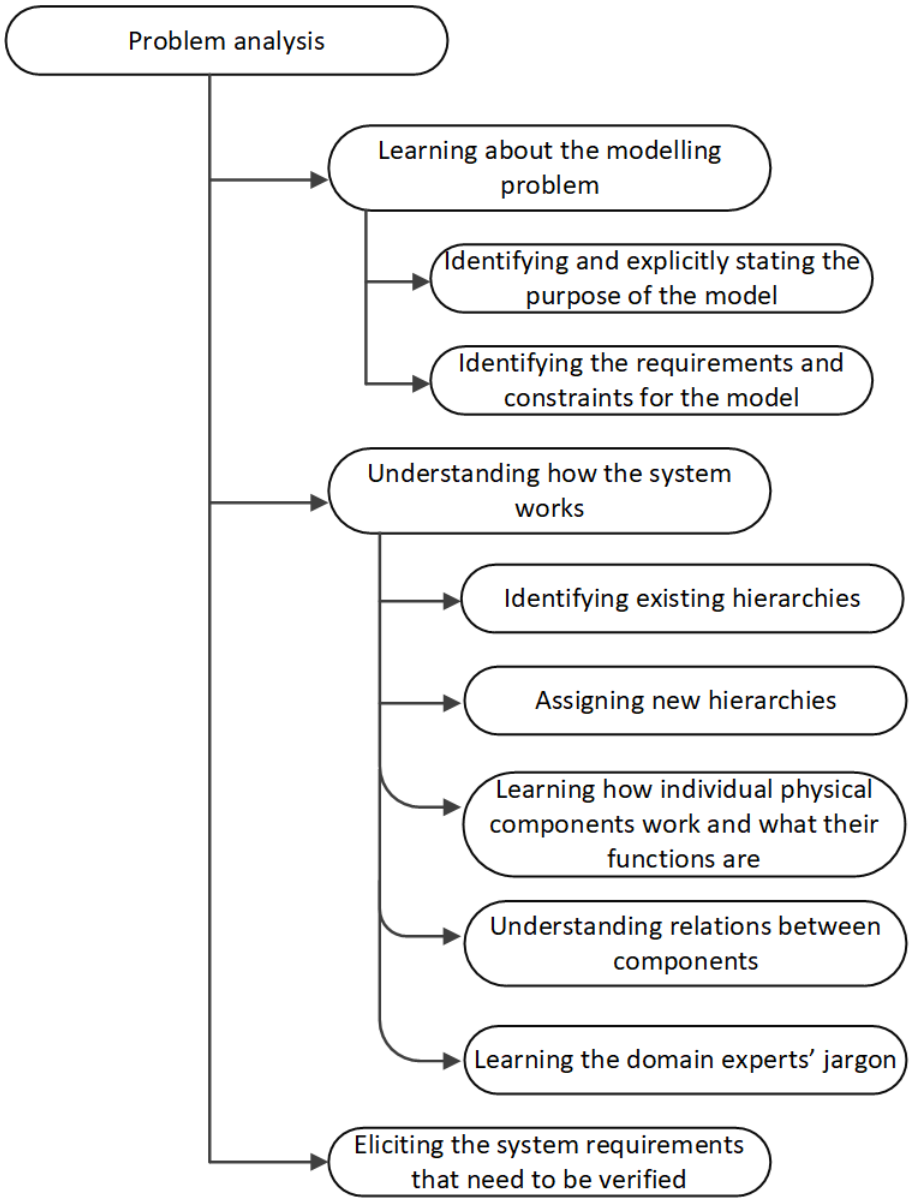


Figure 10.4: Activities that are part of problem analysis during model design.

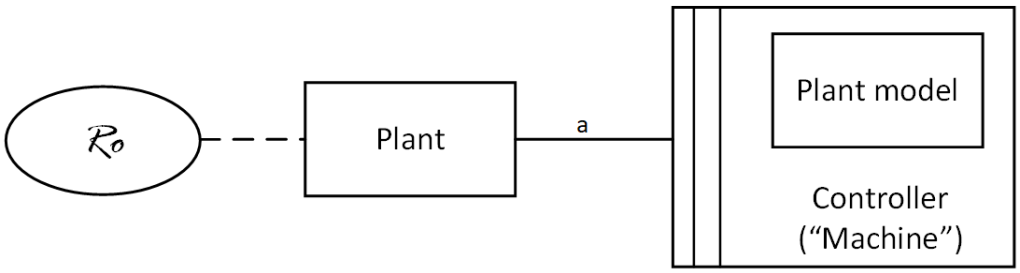


Figure 10.5: The adapted conceptual model to show the aspect of representing the plant in the controller.

Modelling process - Phase 2 - Model design

We start this phase with the clear statement of the model purpose, and requirements for the model itself. The output of this phase is the model, with the documented vocabulary of the terms that are used in the model. In this phase we recognise two main sub-phases. One is concerned with further decomposing (and recomposing) the modelling problem while further learning about the system. The second sub-phase reflects the taxonomy of modelling steps. We created this taxonomy starting from model-system relations described in (Wieringa 1996) and conceptual analysis presented in Chapter 5.

The modelling design steps are shown in Fig. 10.6. The greyed blocks showed the steps we added during our case studies.

One good practice is, for model understandability, to mimic the system in the model by mapping its components or processes one-to-one to the model. This helps when validating if the model is correct and when explaining the model. For example, in the case study of the sorter in Chapter 6, we mimicked the behaviour of physical components - Belt, Entrance tray, Exit tray. Or, in Chapter 8, in the model of the printer we mimicked the processes in the model (the individual maintenance processes and the printing process).

Ideally the model decomposition matches closely with the system decomposition. However, the exact one-to-one mapping is often not possible, and there are cases when due to the formalism constraints, some analogies have to be created. What does not fall into analogy, we name the modelling "trick" to solve the inability to exactly map a certain system or environment phenomenon in the model. Explaining the analogies and these modelling "tricks" becomes important when explaining the model for the purpose of its validation.

Modelling process - Phase 3 - Model justification

The process to follow in model justification is shown in Fig. 10.7. We introduced it in Chapter 5 and validated it and extended in cases described in Chapters 6-8. The grey blocks show the added steps in comparison to the initial method presented in Chapter 5.

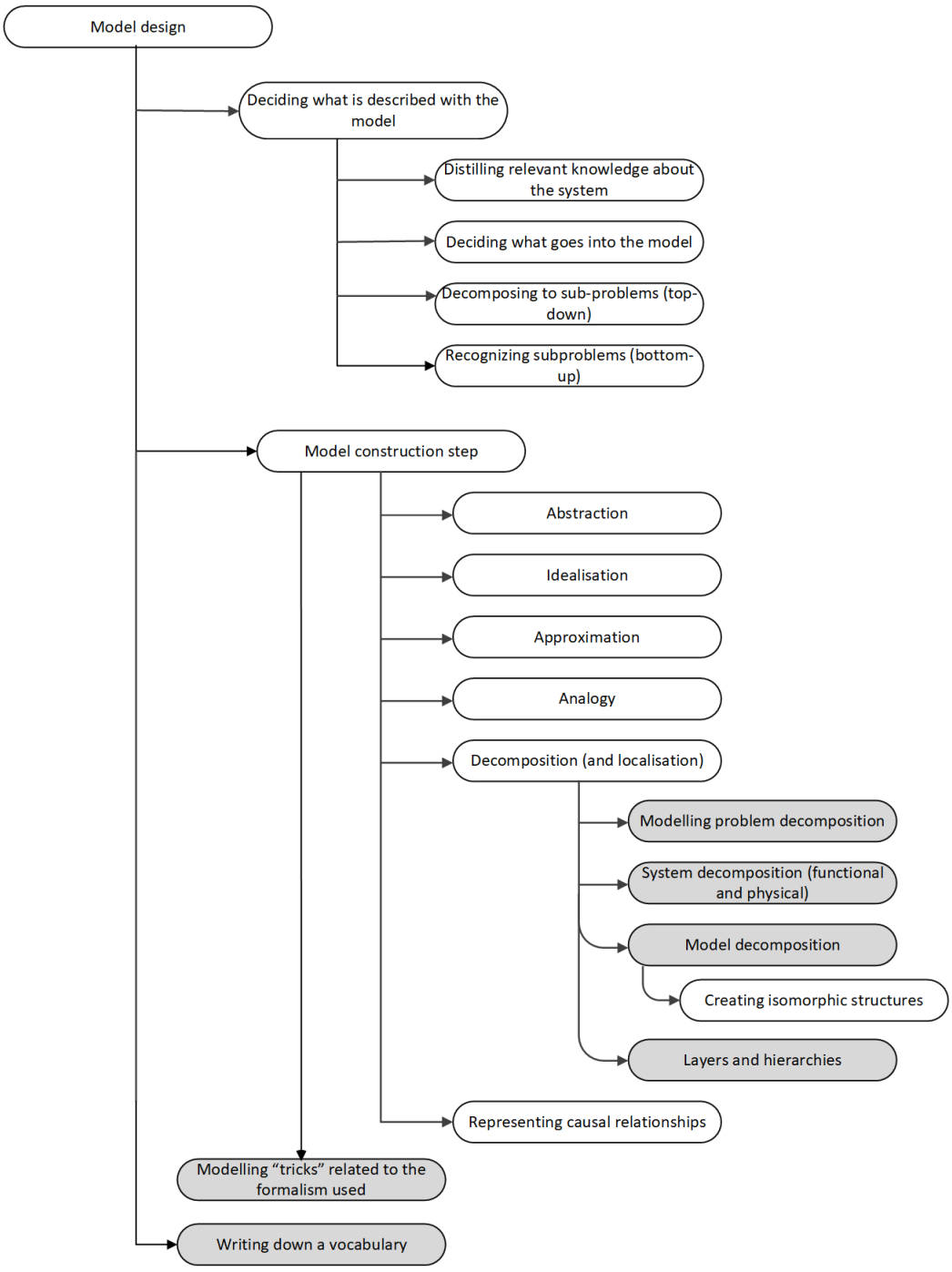


Figure 10.6: Modelling process - Phase 2 - Model design.

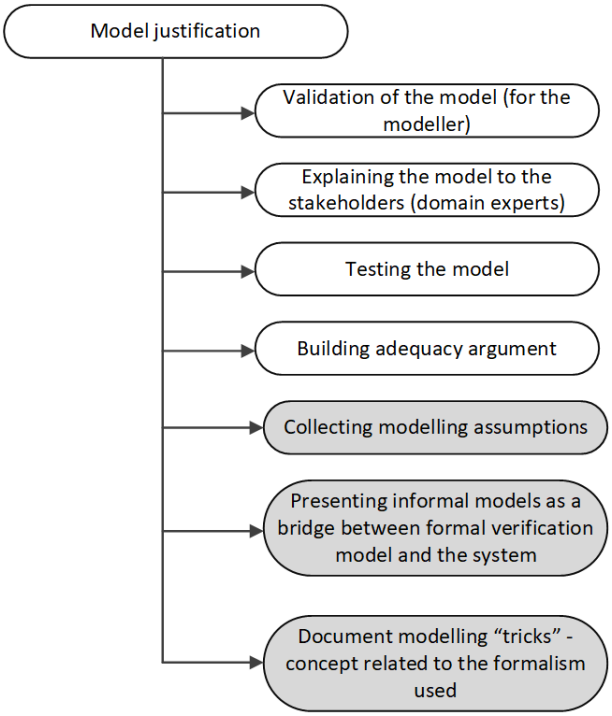


Figure 10.7: Modelling process - Phase 3 - Model justification.

The first step mentions the model validation. Here we refer to the fact that there are many modelling tool suites that have simulation tool built in, so that the modeller can validate if they are designing what they think they are designing.

The second step is about explaining the models. This is important, because the modeller represents *their understanding* of the system. It is important to check with the stakeholders if that understanding is correct. One way is to use the taxonomy of modelling steps or system-model relations (Fig. 10.8), to explain what is in the model and to validate these steps.

Typically in practice, the domain experts provide many details about the system. These details may be relevant for what they are designing, but they are not necessarily relevant for the result of the model with the defined purpose. The taxonomy of modelling steps provides the language to use to explain why some of these details are not in the model. Also, to check if it was justifiable to abstract these details away. For example, in the case in Chapter 7 we abstracted away mechanical details about the rollers moving the papers in the inserter machine. We initially understood that their speed profiles, stiffness and so on, were irrelevant. However, after explaining the model, we learned that there is a slip that happens when moving the papers. This slip is described by mechanical engineers in lots of details. For our model, we approximated it to a certain

value of the paper slip, constant for all the papers. This is good enough for the modelling results of this model. For other models, the slip is described more precisely.

In the same case another example of a modelling "trick" is placing the sensors on different places than where they are in the plant. This does not invalidate the modelling result, but reduces the complexity of the model. It is necessary to explain to the domain experts how the model is still adequate.

The problem when explaining the model to the stakeholders occurs if they do not understand the modelling formalism and language. Or, even worse, if they think they understand it based on the similarity with another formalism they speak. In this case, we propose to use intermediary models, as sketched in Figure 10.9. The risk to remember is that informal models are not a correct representation of the formal ones.

An example of how we modelled it is shown in Chapter 6, where we designed the state machines in parallel with Uppaal models. In the validation of this case we were showing the model to the colleagues who understand the first, but not the latter. Another example is shown in Chapter 8. One of our main stakeholders did not understand Uppaal, but understood the concept of state machines. We used the model architecture diagrams to explain the model, as well as the simulation tool of Uppaal to explain certain scenarios in the model.

The final element in model justification is the collection of modelling assumptions. We propose to create a table with the assumption categories, similar to those we show in Chapter 9, to gather all the relevant assumptions and check them with the stakeholders. The list of verified assumptions is, together with the model the output of this phase.

The classes of assumptions we found sufficient in our cases are as follows.

Modelling assumptions Finally, the modelling assumptions are explicitly addressed and classified into the following categories.

- C1: Assumptions about the system components (processes, physical components, functions... any system based decomposition)
- C2: Assumptions about system aspects
- C3: Constraints on the Plant and Embedded System Environment
- C4: Necessary Assumptions
- C5: Contingent Assumptions
- C6: Assumptions made while co-modelling

They are also shown in Figure 10.11. These classes are on a "meta"-level higher than the assumptions used in concrete cases. This categorisation serves to guide the modeller in looking for assumptions for the concrete system. So, for example the class that refers to system components becomes the assumption about the belt in the case in which we have a system with a conveyor belt.

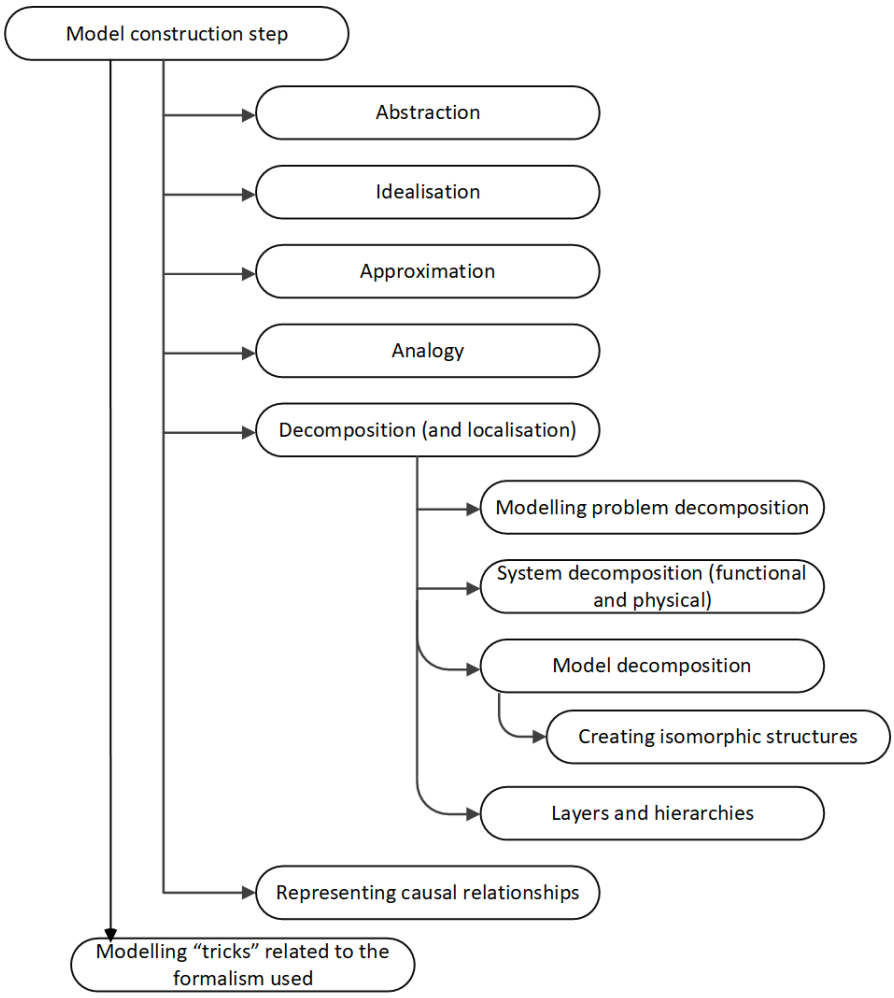


Figure 10.8: Taxonomy of model construction steps.

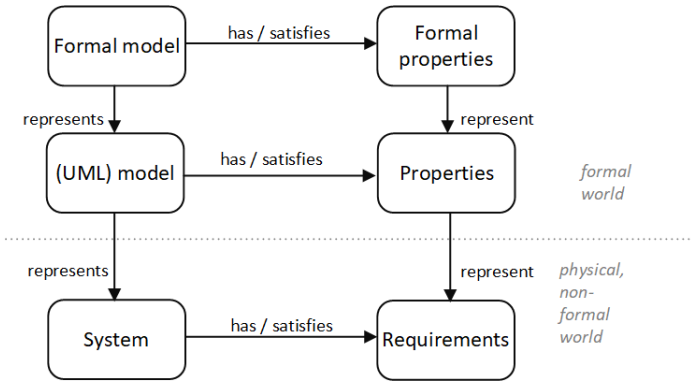


Figure 10.9: Intermediary models are used to explain the formal model.

Correctness argument The model is justified by building the following argument:
((*Model* **satisfies** P) $\wedge$ A) **implies** (*System* **satisfies** R), where
P represents model properties, A represents modelling assumptions, R represents the system requirements.

10.2.3 Back to the higher goal and the right side of the V-model

Our argument that the specified requirement for the specified system segment holds is an input to a higher argument of other requirements being verified. These serve as a basis of an argument that the user top-level goals are satisfied when using the system.

10.2.4 Practical aspect of our method

The formal verification tool suite has its own structure to store the formal verification results. The non-formal parts, the tables of assumptions for example, we recommend to store in a document. In the MBSE domain, the need is recognised to integrate the results of all the models in one place. Ideally, there is an ontology that integrates all these results. This is beyond the scope of this thesis, but we recognise it as an important aspect if this method is to be used in practice.

10.3 To what systems does this method apply

We have at the beginning of this work, already scoped to the following

- Mechatronic systems with electro-mechanical parts
- Workpieces going through the system

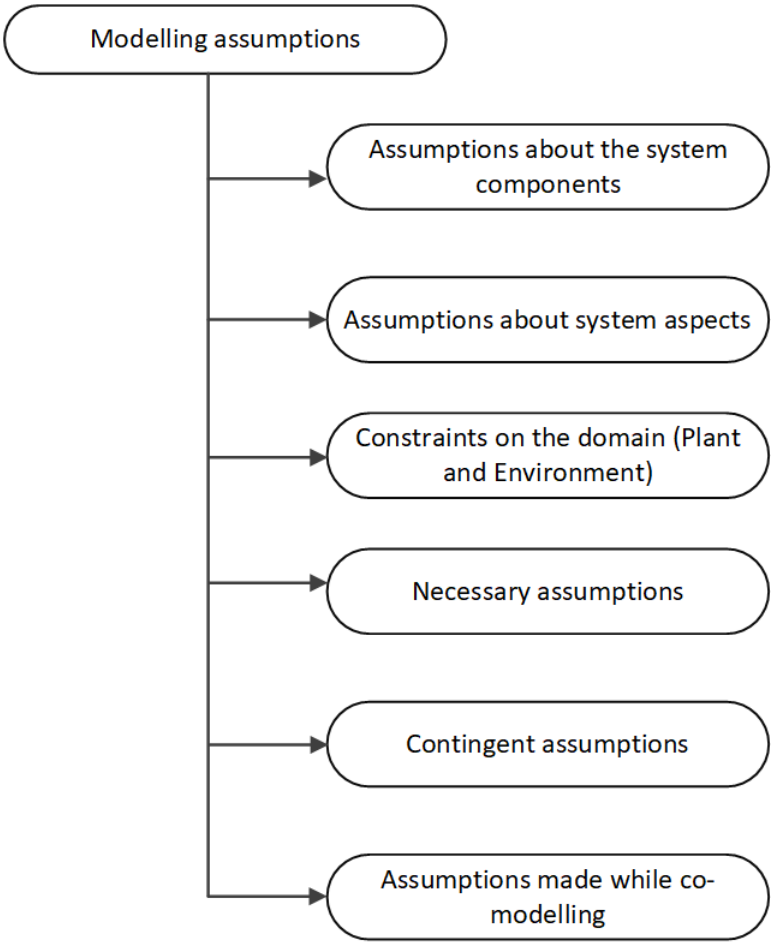


Figure 10.10: Classes of assumptions.

- Modelling and verifying high-level process controller
- Behaviour aspect using states/automata

Also, in our method we assume that the Plant is known. In reality, in early system design phases *both Plant and the Controller* are being designed at the same time. In our method, the exploration of different solutions for the Plant is not visible, we start with one design decision for the Plant, when verifying the requirement. So this places us further down in the V-model of verifying the system requirements.

In his book on methodologies in design science, (Wieringa 2014) discusses generalisation from case studies used as research method. He explains that, unlike in sample-based research that has a clear definition of a population, cases

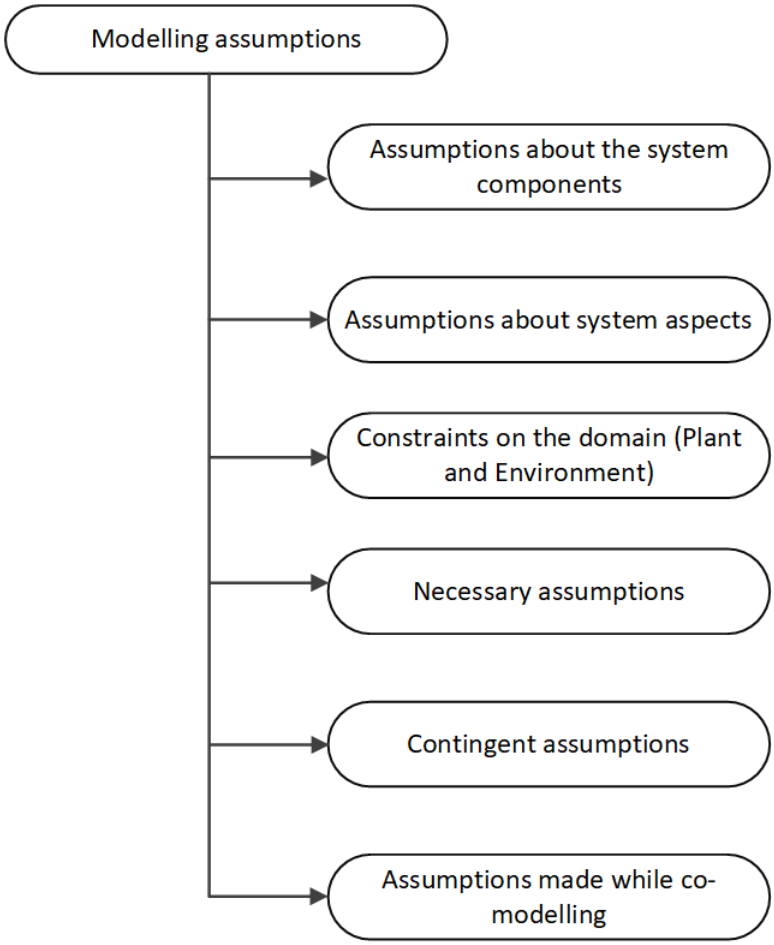


Figure 10.11: Classes of assumptions.

are selected based on similarity. We use the argument of structural similarity to argue that the method can be applied to similar systems. But we can imagine this method being used for example for the systems that do not manipulate workpieces, or those that have some other prominent aspects, for example chemical engineering ones. It is the topic of further work to check whether this is the case.

10.4 Summary

In this chapter, we presented the method overview, consisting of the (1) modelling process, (2) main structures to reason about the artefacts we are designing, and (3) taxonomy and classifications that support individual steps.

As previously stated, the main goal is to build a strong argument that the system requirements hold, reducing the risk of having the wrong model or solving the wrong problem(Hall & Rapanotti 2008b). The main artefact we are building is the model but it is also important to collect modelling assumptions, as their correctness is a precondition for our statement that the system requirements hold. By applying these frameworks, processes, and checklists, the confidence in the model's validity is increased. The quality of the model is also explicitly addressed this way.

Furthermore, we also propose to document the vocabulary used, and an explicit statement what the purpose of the model is. The latter is important to position us in a bigger process of verifying the requirements of the whole system.

Chapter 11

Conclusion and discussion

The main premise of this research is that making informal implicit steps of process of modelling explicit, helps create better, more trustworthy models. We designed a method for structuring informal modelling steps while (1) constructing the model and (2) for model justification.

Our research has been structured around two main design problems and related knowledge questions. We presented them in Chapter 1 and show again here in Table 11.1. The result is the method consisting of conceptual models (structures) and steps to follow when designing a model, presented in Chapter 10.

We started our research with the observation that existing modelling languages and tools do not focus on non-formal modelling steps, which belong more to methodological aspect of modelling; as a result, the quality of a model depends heavily on the skills of its designer. Before doing something about it, we first needed to fully understand the problem. Our initial investigation aimed at answering the following questions: "What exactly *are* the non-formal elements?", "Which of them are recognised and treated by other methods, and which not?", "Why are non-formal elements not treated by some methods?" Finding the answers and subsequently refining the problem into sub-problems facilitated the solution design. In this initial phase we performed a literature survey (Chapter 4), talked to experts, performed conceptual analysis and conducted an initial case study.

The purpose of the literature survey in this phase (Chapter 4) was to explore whether some of the methodological aspects of modelling are covered by researchers and practitioners. Furthermore, we needed to learn (1) how other researchers analyse the modelling process and its non-formal elements. (2) how other modelling methods treat these elements and (3) what elements are not treated. Answering these questions helped clarifying the problem and narrowing down our own questions and goals. We looked at how the different subdisciplines in the software and system engineering answer to the questions we listed above and how they guide the modelling process. We found that some researchers are already working on these problems, by formalising the design

Design problems, conceptual problems and knowledge questions
1.1 Design problem (addressed in **Chapters 5, 6 and 7**) Design a method that guides the **construction** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model. **1.2 Knowledge question** (addressed in **Chapter 10**) To what type of cyber-physical systems is this method applicable?
2.1 Design problem (addressed in **Chapters 5 - 7 and 8 - 9**) Design a method that guides the **justification** of formal models of cyber-physical systems - that is independent of the formalism and - that supports communication among different domain experts about the system and its model. **2.1.1 Design sub-problem** (addressed in **Chapters 9 and 10**) How to organise justifications to facilitate communication among different domain experts?

Table 11.1: The top level design problem is decomposed into design problems, conceptual problems and knowledge questions.

process, and in the MDSE community there is a call to start research on modelling as a process, and even to introduce modelling as a sub-discipline on its own. We aim to contribute to this goal with this thesis. We also looked at how philosophy of science treats models and modelling, and based on the workshop with philosophers and with findings in the literature, we designed the taxonomy of modelling steps.

Our initial version of the method is presented in Chapter 5 and Chapter 6. We designed this initial method by conducting a case in our lab and at the same time performing conceptual analysis. We used this method in the next cases, presented in Chapters 7-9.

In Chapter 3 we describe how we designed our research. We used a combination of research methods, and after designing the initial version of our method, we validated it by using it in an industrial setting. We performed action research, close to the practitioners. We used and built our method on high-tech, cyber-physical systems, which limits the scope of the method to these kinds of systems. The nature of industrial cases provided realistic systems with realistic

problems, but in this setting the ways to validate our method are limited to talking to experts, observing them, working together, and performing interviews. In industrial setting, practitioners have their goals to make and deliver products on time. It is not possible, there is no budget that would allow to conduct experiments to compare modelling with a group using the method and a group not using it. There would be too many uncontrollable, but relevant variables to investigate this with such randomised controlled experiments. Also, in such research we, as researchers, had limited budget (time). Generalisation from case studies is always uncertain, and the most we can say is that in similar cases, it is likely that using this method will yield similar results, without being able to quantify this likelihood.

11.1 Future work

As we mentioned in related work, the community of software modelling has been more vocal in expressing the need to address elements of modelling that are not purely technical. There are initiatives that formulate what would it mean to introduce modelling as a discipline on its own. Some of the steps we cover in our method are mentioned by other researchers as well. It would be beneficial to unify all these approaches into a mature discipline of modelling, with standard names for standard steps and accompanying examples.

Once there is a theoretical framework of modelling as a discipline, the steps we propose, together with those of others, could become part of the existing tool suites. An ontology of assumptions and modelling steps may help there. Furthermore, many tools are getting enhanced by digital "helpers" to guide the process of modelling, so the steps we propose could become a checklist of such a digital helper.

Bucchiarone et al. (2020) talk about the grand challenges in modelling, and as an open challenge they outline the usage of AI in modelling. Precisely formulated modelling steps could contribute helping in conquering this challenge.

Another direction is to relate this work with the work of system architecting as it also structures the plant that together with the software satisfies the requirements. There as well, we can talk about abstractions, idealisations and other modelling steps and discuss and document why the model is correct for the purpose.

Finally, we hope to contribute in educating future model designers. Converting our findings in educational material could help to build modelling expertise. Experience is needed to become a good modeller, but with our method, perhaps we can make the learning process faster.

Bibliography

20-sim (2023). 20-sim tutorial, https://www.20sim.com.

Apostel, L. (1960). Towards the formal study of models in the non-formal sciences, *Synthese* **12**(2): 125–161.
URL: *https://doi.org/10.1007/BF00485092*

Avison, D. E., Lau, F., Myers, M. D. & Nielsen, P. A. (1999). Action research, *Commun. ACM* **42**(1): 94–97.

Baier, C. & Katoen, J. (2008). *Principles of model checking*, MIT Press.

Baskerville, R. L. & Wood-Harper, A. T. (1996). A critical perspective on action research as a method for information systems research, *Journal of Information Technology (Routledge, Ltd.)* **11**(3): 235–246.

Bechtel, W. & Richardson, R. (2010). *Discovering Complexity: Decomposition and Localization as Strategies in Scientific Research*, The MIT Press.

Behrmann, G., David, A. & Larsen, K. G. (2004). *A Tutorial on Uppaal*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–236.
URL: *https://doi.org/10.1007/978-3-540-30080-9*$_7$

Bernardeschi, C., Dini, P., Domenici, A. & Palmieri, M. (2020). Formal verification and co-simulation in the design of a synchronous motor control algorithm, *Energies* **13**: 4057.

Bonnema, G. & Broenink, J. (2016). Thinking tracks for multidisciplinary system design, *Systems* **4**(4).

Bonnema, G. M. (2018). A system design reference model - based on design processes and designers' experience, *J. Integr. Des. Process. Sci.* **22**(1): 39–60.

Booch, G. (2005). *The Unified Modeling Language User Guide*, The Addison-Wesley object technology series, Pearson Education.
URL: *https://books.google.nl/books?id=xfQ8JCbxDK8C*

Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J. & Houston, K. (2007). *Object-oriented analysis and design with applications, third edition*, Addison-Wesley Professional, Boston, MA, USA.

Boon, M. (2006). How science is applied in technology, *International Studies in the Philosophy of Science* **20**(1): 27–47.
URL: *http://www.tandfonline.com/doi/abs/10.1080/02698590600640992*

Boon, M. (2020). Scientific methodology in the engineering sciences, *in* N. D. Diane P. Michelfelder (ed.), *The Routledge Handbook of the Philosophy of Engineering*, Taylor and Francis.

Boon, M. & Knuuttila, T. (2009). Models as epistemic tools in engineering sciences, *in* A. Meijers (ed.), *Philosophy of Technology and Engineering Sciences*, Handbook of the Philosophy of Science, North-Holland, Amsterdam, pp. 693–726.
URL: *https://www.sciencedirect.com/science/article/pii/B9780444516671500306*

Bourguiba, I. & Moa, B. (2012). Improving the semantics of the software cost reduction method, *Journal of Applied Logic* **10**(4): 410–418. Selected papers from the 6th International Conference on Soft Computing Models in Industrial and Environmental Applications.
URL: *https://www.sciencedirect.com/science/article/pii/S1570868312000584*

Bowen, J. P., Butler, R. W., Dill, D. L., Glass, R. L., Gries, D., Hall, A., Hinchey, M. G., Holloway, C. M., Jackson, D., Jones, C. B., Lutz, M. J., Parnas, D. L., Rushby, J. M., Wing, J. M. & Zave, P. (1996). An invitation to formal methods, *IEEE Computer* **29**(4): 16–30.

Bowen, J. P. & Hinchey, M. G. (1995). Seven more myths of formal methods, *IEEE Softw.* **12**(4): 34–41.

Box, G. E. P. (1979). Robustness in the strategy of scientific model building, *MRC technical summary report* .

Brambilla, M., Cabot, J. & Wimmer, M. (2017). *Model-Driven Software Engineering in Practice, Second Edition*, Synthesis Lectures on Software Engineering (SLSE), Springer Cham.

Broenink, J. (2010). Design support and tooling for dependable embedded control systems.

Bruel, J.-M. (1998). Integrating informal and formal specification techniques. why? how? (panel session), *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, Boca Raton/FL, USA*, IEEE CS Press.

Bucchiarone, A., Cabot, J., Paige, R. F. & Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research, *Software and Systems Modeling* **19**(1): 5–13.
URL: *https://doi.org/10.1007/s10270-019-00773-6*

Busari, S. & Letier, E. (2017). Scalability analysis of the radar decision support tool.

Cabot, J. & Vallecillo, A. (2022). Modeling should be an independent scientific discipline, *Software and Systems Modeling* **21**(6): 2101–2107.
URL: *https://doi.org/10.1007/s10270-022-01035-8*

Canon Production Printing (2022). https://cpp.canon. Accessed: 2022-08-25.

Cartwright, N. (1983). *How the laws of physics lie*, Clarendon Press.

Chakravartty, A. (2001). The semantic or model-theoretic view of theories and scientific realism, *Synthese* **127**(3): 325–345.

Chebanyuk, E. & Markov, K. (2016). Model of problem domain "model-driven architecture formal methods and approaches, *International Journal "Information Content and Processing* **2**(3): 202–222.

Choppy, C. & Reggio, G. (2004). Using uml for problem frame oriented software development.
URL: *citeseer.ist.psu.edu/choppy04using.html*

Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., Paige, R. F., Pierantonio, A., Rensink, A., Salay, R., Taentzer, G., Vallecillo, A. & Wimmer, M. (2018). Towards a body of knowledge for model-based software engineering, *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '18, Association for Computing Machinery, New York, NY, USA, pp. 82–89.
URL: *https://doi.org/10.1145/3270112.3270121*

Clark, T., van den Brand, M., Combemale, B. & Rumpe, B. (2015). *Conceptual Model of the Globalization for Domain-Specific Languages*, Springer International Publishing, Cham, pp. 7–20.
URL: *https://doi.org/10.1007/978-3-319-26172-0*$_2$

Clarke, E. M., Klieber, W., Nováček, M. & Zuliani, P. (2012). *Model Checking and the State Explosion Problem*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–30.
URL: *https://doi.org/10.1007/978-3-642-35746-6*$_1$

Coad, P., North, D. & Mayfield, M. (1995). *Object models: strategies, patterns, applications*, Yourdon Press, Upper Saddle River, NJ, USA.

Colburn, T. & Shute, G. (2007). Abstraction in computer science, *Minds and Machines* **17**(2): 169–184.
URL: *http://dl.acm.org/citation.cfm?id=1285757.1285760*

Cross, N. (2001). Design cognition: results from protocol and other empirical studies of design activity, *in* C. Eastman, W. McCracken & W. Newstetter (eds), *Design Knowing and Learning: Cognition in Design Education*, Elsevier Science, Amsterdam, The Netherlands, pp. 79–103.

Dasgupta, S. (1991). *Design theory and computer science: processes and methodology of computer systems design*, Cambridge University Press, New York, NY, USA.

David D. Walden, Garry J. Roedler, K. J. F. R. D. H. T. M. S. (ed.) (2015). *Systems engineering handbook*, Wiley.

Davison, R. M., Martinsons, M. G. & Kock, N. (2004). Principles of canonical action research, *Inf. Syst. J.* **14**(1): 65–.

De Saqui-Sannes, P., Vingerhoeds, R. A., Garion, C. & Thirioux, X. (2022). A taxonomy of mbse approaches by languages, tools and methods, *IEEE Access* **10**: 120936–120950.

Delima, R., Wardoyo, R. & Mustofa, K. (2021). Goal-oriented requirements engineering: State of the art and research trend, *JUITA : Jurnal Informatika* **9**(1).

Douglass, B. P. (2016). *SysML Introduction*, Morgan Kaufmann, pp. 85–145.

Engen, S., Muller, G. & Falk, K. (2022). Conceptual modeling to support system-level decision-making: An industrial case study from the norwegian energy domain, *Systems Engineering* .

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*, number isbn=9780132181273, Pearson Education.

Felderer, M., Gurov, D., Huisman, M., Lisper, B. & Schlick, R. (2018). Formal methods in industrial practice - bridging the gap (track summary), *in* T. Margaria & B. Steffen (eds), *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Springer International Publishing, Cham, pp. 77–81.

Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. M. (1995). *Design Patterns:Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, USA.

Giere, R. N. (2004). How models are used to represent reality, *Philosophy of Science* **71**(5): 742–752.

Giorgini, P., Mylopoulos, J. & Sebastiani, R. (2005). Goal-oriented requirements analysis and reasoning in the tropos methodology, *Eng. Appl. of AI* **18**(2): 159–171.

Gomm, R., Hammersley, M. & Foster, P. (eds) (2000.). *Case Study Method, Key Issues, Key Texts*, SAGE Publications.

Grobelna, I. (2020). Formal verification of control modules in cyber-physical systems, *Sensors* **20**(18).
URL: *https://www.mdpi.com/1424-8220/20/18/5154*

Gunter, C., Gunter, E., Jackson, M. & Zave, P. (2000). A reference model for requirements and specifications, *IEEE Software* **17**(3): 37–43.

Hall, A. (1990). Seven myths of formal methods, *IEEE Software* **7**(5): 11–19.

Hall, A. (1998). What does industry need from formal specification techniques?, *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pp. 2–7.

Hall, A. (2005). Making formal methods work, *SEFM*, pp. 261–262.

Hall, A. (2007). Realising the benefits of formal methods, *J. UCS* **13**(5): 669–678.

Hall, J. G., Mannering, D. & Rapanotti, L. (2007). Arguing safety with problem oriented software engineering, *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pp. 23–32.

Hall, J. G. & Rapanotti, L. (2008a). Assurance-driven design, *2008 The Third International Conference on Software Engineering Advances*, pp. 379–388.

Hall, J. G. & Rapanotti, L. (2008b). The discipline of natural design, *Proceedings of the Design Research Society Conference 2008, 16-19 July, 2008, Sheffield, UK*, Sheffield Hallam University Research Archive.
URL: *http://shura.shu.ac.uk/id/eprint/460*

Hall, J. G., Rapanotti, L. & Jackson, M. (2005). Problem frame semantics for software development, *Software & Systems Modeling* **4**(2): 189–198.
URL: *https://doi.org/10.1007/s10270-004-0062-1*

Hall, J. G., Rapanotti, L. & Jackson, M. (2007). Problem oriented software engineering: A design-theoretic framework for software engineering, *sefm* **0**: 15–24.

Hall, J. & Rapanotti, L. (2016). A design theory for software engineering.

Hall, J. & Rapanotti, L. (2017). A design theory for software engineering, *Information and Software Technology* **87**.

Harel, D. (1987). Statecharts: A visual formalism for complex systems, *Science of Computer Programming* **8**(3): 231–274.
URL: *citeseer.ist.psu.edu/harel87statecharts.html*

Harel, D. & Pnueli, A. (1985). On the development of reactive systems, *in* K. Apt (ed.), *Logics and Models of Concurrent Systems*, Springer, pp. 477–498. NATO ASI Series.

Haveman, S. (2015). *COMBOS: communicating behavior of systems: incorporating simulations in conceptual system design*, PhD thesis, University of Twente, Netherlands.

Heisel, M. (1998). Agendas – a concept to guide software development activites, *in* R. N. Horspool (ed.), *Proc. Systems Implementation 2000*, Chapman & Hall London, pp. 19–32.

Heisel, M. & Souquières, J. (1999). A method for requirements elicitation and formal specification, *in* J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau & E. Métais (eds), *Proc. of the 18th International Conference on Conceptual Modeling*, Vol. 1728 of *Lecture Notes in Computer Science*, Springer, pp. 309–324.

Heitmeyer, C. & Bharadwaj, R. (2000). Applying the scr requirements method to the light control case study, *j-jucs* **6**(7): 650–678.
URL: *http://www.jucs.org/jucs_6_7/applying_the_scr_requirements*

Heitmeyer, C. L., Jeffords, R. D. & Labaw, B. G. (1996). Automated consistency checking of requirements specifications, *ACM Trans. Softw. Eng. Methodol.* **5**(3): 231–261.

Henzinger, T. A. & Sifakis, J. (2007). The discipline of embedded systems design, *IEEE Computer* **40**(10): 32–40.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design science in information systems research, *MIS Quarterly* **28**(1).

Horkoff, J., Aydemir, F. B., Cardoso, E., Li, T., Maté, A., Paja, E., Salnitri, M., Piras, L., Mylopoulos, J. & Giorgini, P. (2019). Goal-oriented requirements engineering: an extended systematic mapping study, *Requirements Engineering* **24**(2): 133–160.
URL: *https://doi.org/10.1007/s00766-017-0280-z*

Hutchinson, J., Whittle, J. & Rouncefield, M. (2014). Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure, *Science of Computer Programming* **89**: 144–161. Special issue on Success Stories in Model Driven Engineering.
URL: *https://www.sciencedirect.com/science/article/pii/S0167642313000786*

INCOSE (2023). The international council on systems engineering, https://www.incose.org/. Accessed 2023-07-02.

Jackson, D. (2021). *The Essence of Software: Why Concepts Matter for Great Design*, Princeton University Press.

Jackson, M. (1995). *Software Requirements and Specifications: A lexicon of practice, principles and prejudices*, Addison-Wesley.

Jackson, M. (2000). *Problem Frames: Analysing and Structuring Software Development Problems*, Addison-Wesley.

Jackson, M. (2009). Some notes on models and modelling, *in* A. Borgida, V. Chaudhri, P. Giorgini & E. Yu (eds), *Conceptual Modeling: Foundations and Applications*, Vol. 5600 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 68–81.

John Fitzgerald, Peter Gorm Larsen, M. V. (ed.) (2016). *Collaborative Design for Embedded Systems*, Springer Berlin, Heidelberg.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., Emam, K. E. & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering, *IEEE Trans. Softw. Eng.* **28**: 721–734.
URL: *http://portal.acm.org/citation.cfm?id=636196.636197*

Kitchenham, B., Pickard, L. & Pfleeger, S. L. (1995). Case studies for method and tool evaluation, *IEEE Software* **12**(4): 52–62.

Knuuttila, T. & Boon, M. (2011). How do models give us knowledge? the case of carnot's ideal heat engine, *European Journal for Philosophy of Science* **1**(3): 309.
URL: *https://doi.org/10.1007/s13194-011-0029-3*

LabView (2023). http://www.ni.com/labview/. Aceesed 2023.

Larsen, K. G., Pettersson, P. & Yi, W. (1997). Uppaal in a nutshell, *International Journal on Software Tools for Technology Transfer* **1**(1): 134–152.
URL: *https://doi.org/10.1007/s100090050010*

Laymon, R. (1989). Applying idealized scientific theories to engineering, *Synthese* **81**(3): 353–371.

Laymon, R. (1990). Computer simulations, idealizations and approximations, *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association* **1990**: 519–534.

Lee, A. S. & Baskerville, R. (2003). Generalizing generalizability in information systems research, *Information Systems Research* **14**(3): 221–243.

Li, Z., Hall, J. G. & Rapanotti, L. (2014). On the systematic transformation of requirements to specifications, *Requirements Engineering* **19**(4): 397–419.
URL: *https://doi.org/10.1007/s00766-013-0173-8*

Liebel, G., Marko, N., Tichy, M., Leitner, A. & Hansson, J. (2018). Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice, *Softw. Syst. Model.* **17**(1): 91–113.
URL: *https://doi.org/10.1007/s10270-016-0523-3*

Long, D. & Scott, Z. (2011). *A Primer for Model-Based Systems Engineering, 2nd edition*, number ISBN-13: 9781105588105, lulu.com.

Mader, A. (2000). What is the method in applying formal methods to PLC applications?, *4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM)*, Shaker Verlag, pp. 165–171. http://www.cs.utwente.nl/ mader/PAPERS/method.ps.gz.

Mannering, D. P. (2010). *Problem Oriented Engineering for Software Safety*, PhD thesis, The Open University, UK.

Maurice Heemels, G. M. (ed.) (2006). *Boderc: Model-based design of high-tech systems*, Embedded Systems Institute, Eindhoven, The Netherlands.
URL: *https://a.storyblok.com/f/74249/x/df766fe6b2/bodercbook.pdf*

McMullin, E. (1985). Galilean idealization, *Studies in History and Philosophy of Science* **16**: 247–273.

Merriam-Webster dictionary (2018). https://www.merriam-webster.com/dictionary/assumption.

Mhenni, F., Choley, J.-Y., Penas, O., Plateaux, R. & Hammadi, M. (2014). A sysml-based methodology for mechatronic systems architectural design, *Advanced Engineering Informatics* **28**(3): 218–231. Multiview Modeling for Mechatronic Design.
URL: *https://www.sciencedirect.com/science/article/pii/S1474034614000342*

Muller, G. (2011). *Systems Architecting: A Business Perspective (1st ed.)*, number https://doi.org/10.1201/b11816, CRC Press, Taylor Francis.

Muller, G. (2014). Teaching conceptual modeling at multiple system levels using multiple views, *Procedia CIRP* **21**: 58–63. 24th CIRP Design Conference.
URL: *https://www.sciencedirect.com/science/article/pii/S2212827114007483*

Muller, G. (2020). Research in systems architecting.
URL: *https://www.gaudisite.nl/ArchitectingResearchMethodSlides.pdf*

Murray-Smith, D. (2015). *Testing and Validation of Computer Simulation Models: Principles, Methods and Applications*, number isbn=9783319150987 in *Simulation Foundations, Methods and Applications*, Springer International Publishing.

Neopost Technologies (n.d.). https://www.quadient.com/nl/homepage. Accessed July 2023.

Nguyen, C. M., Sebastiani, R., Giorgini, P. & Mylopoulos, J. (2018). Multi-objective reasoning with constrained goal models, *Requirements Engineering* **23**(2): 189–225.
URL: *https://doi.org/10.1007/s00766-016-0263-5*

Oce Museum (.). https://www.ocemuseum.nl/oce-technologies/printing/?lang=en. Accessed: 2022-08-25.

Parnas, D. & Madey, J. (1995). Functional documents for computer systems, *Science of Computer programming* **25**: 41–61.

Potts, C. (1993). Software-engineering research revisited, *IEEE Softw.* **10**(5): 19–28.

Robinson, S. (2008). Conceptual modelling for simulation part i: definition and requirements, *Journal of the Operational Research Society* **59**(3): 278–290.
URL: *https://doi.org/10.1057/palgrave.jors.2602368*

Robson, S. G., Tangen, J. M. & Searston, R. A. (2021). The effect of expertise, target usefulness and image structure on visual search, *Cognitive Research: Principles and Implications* **6**(1): 16.
URL: *https://doi.org/10.1186/s41235-021-00282-5*

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-oriented modeling and design*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Sandee, J., van den Bosch, P., Heemels, W., Muller, G. & Verhoef, M. (2006). 6.3.1 threads of reasoning: A case study, *INCOSE International Symposium* **16**(1): 895–909.
URL: *https://incose.onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2006.tb02789.x*

Sarhadi, P. & Yousefpour, S. (2015). State of the art: hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software, *International Journal of Dynamics and Control* **3**(4): 470–479.
URL: *https://doi.org/10.1007/s40435-014-0108-3*

Schumann, J. & Goseva-Popstojanova, K. (2019). Verification and validation approaches for model-based software engineering, *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 514–518.

Seater, R., Jackson, D. & Gheyi, R. (2007). Requirement progression in problem frames: deriving specifications from requirements, *Requirements Engineering* **12**(2): 77–102.

Selic, B. (2012). What will it take? a view on adoption of model-based methods in practice, *Software & Systems Modeling* **11**(4): 513–526.
URL: *https://doi.org/10.1007/s10270-012-0261-0*

Simon, H. A. (1996). *The Sciences of the Artificial*, The MIT Press.

Smith, B. C. (1985). The limits of correctness, *SIGCAS Comput. Soc.* **14,15**(1,2,3,4): 18–26.
URL: *https://doi.org/10.1145/379486.379512*

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning, *Cognitive Science* **12**(2): 257–285.

SysML - Open Source Specification Project (2023). http://www.sysml.org/. (Accessed 15 April 2011).

Szyperski, C. (2002). *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley.

Thacker, B. H., Doebling, S. W., Hemez, F. M., Anderson, M. C., Pepin, J. E. & Rodriguez, E. A. (2004). Concepts of model verification and validation.

Thalheim, B. (2010). Towards a theory of conceptual modelling, *J. UCS* **16**: 3102–3137.

Thalheim, B. (2012). The science and art of conceptual modelling, *in* A. Hameurlain, J. Küng, R. Wagner, S. W. Liddle, K.-D. Schewe & X. Zhou (eds), *Transactions on Large-Scale Data- and Knowledge-Centered Systems VI*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 76–105.

Thalheim, B. (2022). Models: the fourth dimension of computer science, *Software and Systems Modeling* **21**(1): 9–18.
URL: *https://doi.org/10.1007/s10270-021-00954-2*

Uppaal (2023). Uppaal, https://uppaal.org. Accessed 2023-07-01.

van Amerongen, J. (2010). *Dynamical Systems for Creative Technology*, Control Products B.V.

van Breemen, A. J. (2001). *Agent-Based Multi-Controller Systems*, PhD thesis, University of Twente.

van Lamsweerde, A. (2009). *Requirements Engineering - From System Goals to UML Models to Software Specifications*, Wiley.

Vernon, V. (2013). *Implementing domain-driven design*, Addison-Wesley.

Vincenti, W. G. (1990). *What engineers know and how they know it: Analytical studies from aeronautical history*, Johns Hopkins University Press, Baltimore.

Weisberg, M. (2007). Three kinds of idealization, *Journal of Philosophy* **104**(12): 639–659.

Whittle, J., Hutchinson, J. & Rouncefield, M. (2014). The state of practice in model-driven engineering, *IEEE Software* **31**(3): 79–85.

Wieringa, R. (2003). *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*, Morgan Kaufmann.

Wieringa, R. (2014). *Design science methodology for information systems and software engineering*, Springer. 10.1007/978-3-662-43839-8.

Wieringa, R. J. (1989). Three roles of conceptual models in information system design and use, *in* E. Falkenberg & P. Lindgreen (eds), *Information System Concepts: An In-dept Analysis*, North Holland, pp. 31–51.

Wieringa, R. J. (1996). *Requirements Engineering: Frameworks for Understanding*, Wiley.

Wieringa, R. J. (2009). Design science as nested problem solving, pp. 1–12.

Woods, J. & Rosales, A. (2010). Virtuous distortion model-based reasoning in science and technology, *Model-Based Reasoning in Science and Technology: Abduction, Logic, and Computational Discovery*, Vol. 314 of *Studies in Computational Intelligence*, Springer Berlin / Heidelberg, pp. 3–30.
URL: *http://www.springerlink.com/content/f66n501256017186*

Wupper, H. & Meijer, H. (1997). A taxonomy for computer science., pp. 217–230.

Xiao, H., Li, Z., Yang, Y., Deng, J. & Wei, S. (2021). An extended metamodel of problem frames for enriching environmental descriptions, *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pp. 428–434.

Yin, R. K. (2003.). *Case Study Research: Design and Methods*, number 5 in *Applied Social Research Methods Series*, 3rd edn, SAGE Publications.

Zave, P. & Jackson, M. (1997). Four dark corners of requirements engineering., *ACM Transactions on Software Engineering Methodology* **6**(1): 1–30.

Appendix A

Appendix: An excerpt from the modelling handbook

Below is an excerpt from the modelling handbook designed in the case described in Chapter 7. (The name of the project is deleted (it says "xxx" here) for confidentiality reasons.)

1 What is the model used for?

This section contains questions to discuss **before and while** modelling. Answers to these questions say something about the purpose of the model, requirements to be verified, and limitations and trade-offs that will have to be made.

The roles involved in answering the questions are:
- System architect
- Verification engineer
- Validation engineer
- Testing engineer
- Modeller
- System integrator

The order of the steps is not as they are listed. Finding answers is an iterative process. It starts before the model design, and it may change during the model design.

1.1 Decide on the purpose of the model. What is it going to be used for?

There is usually more than one purpose a model has to fulfil. One purpose will always be that a model is used as an aid for R&D. Below, some of the possible classes of purposes are listed. In the future there may be more.

Verification and testing of different system parts. (What are we doing with the model?)

- Software testing
    - Testing all possible scenarios of user requests and paper lengths
    - Regression testing of software functionality over different software releases
    - Endurance testing
    - Fault tolerance – how will the software react to faults in the mechanics.
    - Fault tolerance – how will the software react to some irregularities of the material (e.g., different length of the paper)

- Mechanics verification
    - Testing limitations and limits of the specification
    - Testing the impact on the software behaviour after changes in the emulator (model). For example – what will happen if sensor is added?
    - Feedback from software engineers to mechanical engineers

- System verification
    - Regression testing of system functionality over different software releases
    - Endurance test
    - Fault tolerance – reproduce difficult fault scenarios.

    - Testing limits/limitations
    - Visualisation
    - Communication with other models
    - System performance
    - Functional or non-functional requirement

Verification and testing in different project phases. (When are we using the model?)

- Feasibility study – the mechanics does not exist yet or is not stable.
- End of feasibility study – the prototype is there, but a lot of people want to work with it
- Phase 2 – more mechanics are there: regression tests, reproduce difficult fault scenarios – for some things it is easier to test with the model than with the emulator.

Communication (What are we doing with the model?)

- With colleagues
- Departments
- Companies.

Example: In the (xxx) project the model has the following purposes:

- R&D development aid
    - Stable platform
    - Reproduce faults
    - Regression tests

- Verification*
    - Endurance test
    - Testing limits/limitations spec.
    - Regression over releases

- Shortening integration time

* Verification department has set of tests for the acceptance criteria. They are run every day (testing different scenarios with different lengths of paper). These tests are run on the emulator before the prototype is available.)

Answering these questions helps in the next steps, deciding the following:

- What parts of the system to model
- What system components and aspects to leave out (not to model)
- What parts of the system to develop first
- How much time to spend on the model
- Whether to document it and how detailed
- Whether to make it understandable and for whom

1.2 What are the constraints for satisfying model's goals?

Related to model's goals, there are **constraints** that imply some compromises and trade-offs.

Constraints can be:

- Reducing costs,
- Reducing time,
- Reducing number of people involved
- Constraints of the tools, hardware and software resources
- Limited knowledge of tools, software, programming languages
- etc.

Also, one should be aware of short-term and long-term goals. For example, having the current phase finished as soon as possible is a short-term goal and having the processes in the future projects more efficient is a long-term goal.

Different people's work is evaluated in relation to different goals, some of them are evaluated according to the short-term goals and some of them are evaluated against long-term goals. That is why it is good that all stakeholders of the modelling process answer these questions.

1.3 What is the expected model's lifecycle?

The questions to answer here are:

- Will the model be re-used in a future project?
- Will some model parts be re-used in a future project?
- Will the model be maintained over the whole product design and maintenance phase?

Answering these questions helps deciding the following:

- If and how detailed to document the model
- How much attention to pay on understandability of the model
- How much attention to pay on re-usability/evolvability of the model

1.4 What requirements are verified?

Functional or non-functional requirements

Functional requirements are requirements for the system behaviour (they say what the system should do). For example, functional requirements are:

- Paper has to move through the inserter, along the defined paper path
- Paper should be folded in the folder the way the user specified etc.
- Timing requirements

Non-functional requirements do not say anything about the system behaviour, but specify system quality features like security, performance etc.

Example: In (xxx) project the requirements verified are:

- Paper moving along the path in normal working mode
- Paper moving along the path in the load ‘n go mode
- Fault tolerance in case of
    - a broken sensor
    - sensor slip
    - track slip
    - paper jam

Software requirements vs. system requirements

Software requirements define how the software should behave, so the specifications are for the signals on sensors and actuators.

System requirements are the requirements for the whole system and refer to the mechanics and the material.

If the model is used for verification, then how the model is designed, what to put in and what to abstract away, depends on the requirements that will be verified with the model. Therefore, answering these questions helps deciding the following:

- What parts and aspects of the system to describe with the model

If the model is used for testing or simulation, again, the system properties that we want to test determine how to design the model.

1.5 Is it necessary to decompose (split) models into more models for different purposes and requirements?

If there are different purposes and different requirements that we want to test with the emulator, it can happen that the model will contain different system descriptions for different purposes. If, as a result, the model becomes too complex, it makes sense to split it into more models, each for different requirement or purpose.

The disadvantage of splitting is the increase of the maintenance effort.

For example, (xxx) emulator has the above-mentioned purposes and it would take too much time and effort to split the model. Besides, the model is not too complex to maintain and understand even though it serves a few purposes.

1.6 Who will design and who will use the model?

Here are some possible answers to these questions.

Model designer can be

- Someone from the company who has already designed the similar model
- Someone from the company familiar with the model, but didn't design the previous model
- Someone from the company not-familiar with the model
- Someone outside the company

Users of the model can be

- Verification and validation engineer
- Developers, software engineers
- Mechatronics engineers
- Testing engineers
- Mechanical engineers

Example: In the (xxx) project the modeller is from the company and the users are:

- Testing engineer
- Validation and verification engineer
- Mechatronic engineer
- Software engineer

Answering these questions helps deciding the following:

- Estimate the time necessary to make the model
- Whether to document it and how detailed
- To what extent it has to be modularised and understandable

1.7 What are the quality criteria of the model?

These criteria are not independent of each other. There can be a relationship or an inverse relationship between them.

Truthful. Model mimics some aspects and behaviour of the system. It will never be exactly the same as the system, but it has to be good enough to make the results obtained from emulation/verification/testing meaningful.

> For example, in xxx emulator, the track has one slip parameter, no matter how many pinches there are in the real track. Any wear-out of the pinch, or slip of the paper is described with this parameter. This is good enough for the testing needed.

Software independent. The more we want to describe the mechanics independently from the software, the more the model becomes complex. It is always a trade off between complexity and completeness (truthfulness).

The emulator describes the mechanics. Sometimes, for software engineers it is difficult to forget about what the software does and focus *only* on the phenomena and behaviour in the mechanics that reacts to sensors signals and turns on and off the actuators. Also, a modeller should be aware that mechanics does not 'know' things that the software 'knows', like for example the length of the paper and number of papers in a set.

However, if the modeller takes into account *all* possible values and combinations of signals that can come to clutches and motors, the model will become too complex. Sometimes, it makes sense to rely on the fact that the software will control the mechanics in a certain way.

> In (xxx) inserter, BRE and envelope pass through the same path, starting from their feeders to the inserter module.
>
> Theoretically, it is possible to design a control that turns on the clutches in both feeders and takes a BRE and an envelope at the same time. This means that they could 'meet' right after they exit each from their feeder and continue to be guided together.
> However, control (software) will never do that.
>
> Thus, in the model, on the path segment after their two feeders, only arrival of either one of them is possible.

Complete. What goes in the model depends on what we want to test and/or verify and on the model's purpose. All aspects of the system that influence the property we are checking should be present in the model.

(It is actually difficult to know whether the model is complete. After a lot of work with the model, one can only be more convinced it is complete.)

Understandable. This means that one can understand what every part of the model means, what it describes.

This is a subjective criterion so together with it *always* comes the question: "For whom does the model have to be understandable?" Possible answers are:

- To the verification and testing engineer
- To the mechanical engineer
- To the software engineer
- To another modeller

How much it is necessary that the model is understandable depends on answering the questions about the model purpose, about who will use it and what the constraints are. Documentation, intuitive and consistent naming conventions, simplicity of the model should increase understandability. Understandability is good for peer-reviewing, communication with other people and when we want to re-use the model. However, if the model will not be re-used later and if there are tight timing constraints, then writing the documentation and spending more time on naming conventions is not a priority.

Traceable. Wherever possible, it is good to keep the layout of the model that can be mapped to the system structure. This helps to:

(1) Justify modelling decisions, because it is easy to see that what we wanted to describe about the model exists somewhere in the system. If something is changed in the mechanics, it is easy to make changes in the model.
(2) Trace the source of problems in the system if emulator shows an error of a model component.

Re-usable. If small changes in the system require small changes in the model, than he model is re-usable. To have (parts of) the model reusable, it is necessary to estimate what system parts ill be changed and this is not always possible. Also, having the model re-usable may increase its complexity.

Example: In the xxx project the model is traceable – almost all components can be mapped to the system components.

It was the first time that such a model was made, and it was difficult to predict whether it will be useful to re-use it in a different project, so it is not made to be re-usable. Also, the modeller was not encouraged to make the documentation, since it was meant to be used as long as the project is relevant.

Also, understandability was not relevant, since one man handles almost all the necessary job with it.

Maintainable. Good decomposition and traceability help in the model's maintenance. Whether to increase maintainability depends on the model purpose, its users and modelling constraints.

Simple. In the case of state based models, this means that one can keep track of the states and events in the table. This way a modeller can be sure that the model describes what he thinks it describes and not something else.

Example: The (xxx) inserter behaviour is described with state machines. The whole emulator model is a combination of sequential and parallel state machines.

Those components described with parallel sub-state machines, should stay simple enough so that the modeller still can debug and test the model. If there are too many state machines running in parallel, it becomes impossible to trace all possible behaviours.

LabVIEW tool does not have the ability to do model-checking – testing all possible behaviours automatically – so the justification of the model is the modeller's responsibility.

Small. This depends on the modelling language, tools and hardware components we are using.

Example: In the (xxx) project, the number of FPGA board input-outputs was critical, so some of the modelling solutions were chosen to reduce the number of inputs and outputs needed.

Uniformly designed. The model can be designed with already existing modules and with modelling decisions proven to be good.

Often, there are a number of equally suitable modelling solutions to describe one thing. It is easier to maintain the model and to communicate with people via model, if one description is chosen, if there is a uniformity of modelling decisions.

Deciding on priorities for satisfying model criteria helps deciding on the following:

- The amount of resources (time, money, people) to invest in
    - Writing the documentation
    - Uniform notation
    - Naming conventions
    - Writing the rationale of modelling decisions
    - Dividing the model into sub-modules

1.8 What modelling tools and languages will be used?

This decision depends on the following factors:

- What is already accessible in the company?
- What tools modellers already know or like?
- What is the cost of certain development package?
- What formalism describes the mechanics best?

Example:
In the (xxx) project the model is based on the state machine descriptions because they are suitable for describing the paper path.

FPGA model is used because it can describe real-time, parallel behaviour that emulates well the response of sensors and control of actuators.

LabVIEW package was chosen because developers are familiar with it and they find it easier to learn than VHDL. Model is easier to debug and understand in LabVIEW.

Another advantage of the LabVIEW is that it forces people to think about the mechanics and not the software when they are describing the mechanics. SystemC for example compiles for VHDL[1], but the model would look like a program. It had been observed that for software developers it was difficult to stop thinking about what the software does, so in the end LabView was chosen.

Finally, LabVIEW compiles for different targets, so the model made is portable to different hardware platforms.