Model and Dependency Management in Mechatronic Design

AHSAN QAMAR

Doctoral Thesis
Stockholm, Sweden, 2013
Academic thesis, which with the approval of the Royal Institute of Technology, will be presented for public review in fulfillment of the requirements for a Doctorate of Technology in Machine Design. The public review is held in Room B319, Department for Machine Design, Royal Institute of Technology, Brinellvägen 85, Stockholm on 2013-03-27 at 13:00.

© Ahsan Qamar, 2013, Version 20130306204300

Print: US-AB
Abstract

Managing consistency is a major concern in the design of complex engineering systems. At times, inconsistencies may lead to wrong decisions, resulting in design flaws which can compromise safety and cause failures. One cannot forget the 1999 NASA unmanned MARS Climate Orbiter, which was destroyed due to use of inconsistent units by design teams. Sadly, the history of inconsistency causing failures does not end there. In 2006 Airbus suffered a massive 6.1 billion dollar loss due to use of inconsistent specifications in different versions of design tools. So what causes inconsistency, and how best to avoid it? These are some of the critical questions behind the research reported in this thesis.

Today’s engineering systems cannot be designed by a single individual, but require the efforts of design teams each managing a portion of the overall problem. Naturally, information exchange between teams is necessary for effective decision making. However such communication is often error-prone and inadequate to manage dependencies between tasks, operations, components or properties. As a consequence, inconsistencies and design errors arise, which may cause catastrophic failures.

This thesis investigates the nature of dependencies, typically in the design process of mechatronic products, and proposes an approach for model and dependency management. The proposed solution is based on an expressive Domain Specific Language which enables capturing dependencies (between disparate models) formally and explicitly. This language is called the Dependency Modeling Language (DML), and the supporting tool is named the Dependency Modeler. The overall approach is exemplified through a robot design example, where the DML is used to capture dependencies between mechanical design and control design models. In support of the DML, dependency patterns gather known dependency relationships between different types of properties - such as a pattern between system hierarchy and mechanical CAD assembly. Model transformations are essential to support execution of such patterns and to support the necessary information exchange between disparate models to enable dependency modeling. Transformations supporting the dependency pattern between system hierarchy and mechanical CAD assembly are illustrated for the robot example. Initial reflections on the Dependency Modeler show a strong potential to support change management, workflow management and consistency management.

Future work targets further development and testing of DML in order to achieve a sound platform for dependency management. A development environment supported by an integration framework - encompassing different model-based design tools - is envisioned as an infrastructure for model management in mechatronic design. It is hoped that such an infrastructure will equip designers with the best possible tools to make better decisions and to spot design errors that might otherwise be fatal.

**Keywords:** Dependency Modeling, Model-Based Systems Engineering (MBSE), Mechatronic Design, Model Integration, Tool Integration, Common Language, Mechatronic Challenges, Domain Specific Language.
Terminology

The terminologies existing in different engineering disciplines are usually understood and treated differently outside the respective discipline. Due to the multidisciplinary nature of this research, it is vital to be consistent in the use of key terminology. Hence, the reader is urged to have a look at the following terminology before reading the rest of this document. Note that some of terminology is borrowed from other references, while the rest is formulated in accordance with the use in this thesis.

- **Approach**: The method used or steps taken in setting about a task, or a problem (Hubka and Eder, 1988, Chap. Introduction).

- **Aspect**: A particular part or feature of something (Oxford Dictionaries Online, 2012).

- **Consistency**: An absence of contradictions (Herzig et al., 2011), or when something at one place contradicts with something at another place (Finkelstein, 2000).

- **Dependency**: Something that is dependent on something else (Merriam Webster, 2012). In this thesis, dependency refers to a situation when the values of two or more properties affect each other.

- **Design**: The activity of designing (but not the final result) (Hubka and Eder, 1988, Chap. Introduction).

- **Design concept**: The result of a design activity.

- **Discipline**: A technical fields such as electrical engineering, mechanical engineering.

- **Domain**: Perspective with respect to a technical system, e.g., mechanical, electrical, software perspectives.

  - *Mechatronic Product* is a multi-domain product rather than multi-disciplinary product.
– *Mechatronic Design* is a multi-disciplinary activity, but it is partly performed by combining mono-disciplinary efforts.

- **Framework:** What provides the user some guidelines, and special tools to work on a task.

- **Formalism:** A formalism consists of a language, a semantic domain and a semantic mapping function giving meaning to model in the language (Giese et al., 2007).

- **Function:** The duty of a technical system to deliver specified effects at its output (Hubka and Eder, 1988, Chap. Introduction).

- **Infrastructure:** Facilities and systems serving a technical system or a process (e.g., a design process).

- **Inter-disciplinary:** is the same as multi-disciplinary.

- **Interface:** A common boundary between components, sub-systems, systems, and products, e.g., a physical connection point (Oxford Dictionaries Online, 2012).

- **Interaction:** Behavior that takes place over an interface.

- **Level of Abstraction:** Level that decides the amount of different questions that can be answered through a model (Mosterman and Vangheluwe, 2004).

- **Methodology:** What consists of techniques for performing a task, i.e., what specifies how to perform a task (Estefan, 2008).

- **Module:** A component or a group of components with a defined interface and known interaction abilities.

- **Model:** Abstraction of a real-world artifact (Buur and Andreasen, 1989).

- **Process:** A logical sequence of tasks performed to achieve a particular objective, defining what is to be done (Estefan, 2008).

- **Relations:** An aspect or quality that connects two or more things or parts as being, belonging or working together (Oxford Dictionaries Online, 2012).

- **Technical System:** A composite of physical elements and their interactions, which receives inputs, and delivers effects to guide and drive a technical process (Hubka and Eder, 1988, Chap. Introduction).

- **Tool:** An instrument or means that (when applied to a particular method) can enhance the efficiency of a task, provided it is applied properly and by somebody with proper skills and training.
• **Technology:** The interaction of a technical system with a technical process (analogous to the interaction of a tool and a work piece), or how the effects (output) of a technical system or human being act to guide and drive a technical process or an operation within it (Hubka and Eder, 1988, Chap. Introduction)

• **View:** A representation of a whole system from the perspective of a related set of concerns (The Institute of Electrical and Electronic Engineers, 2013a).

• **View-point:** A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis (The Institute of Electrical and Electronic Engineers, 2013a).
Acknowledgements

Graduating with a Master’s degree, I was enthusiastic to pursue a PhD. Needless to say, I was naive of what PhD research entails - it definitely takes a toll on you! However, when I look back to where I began this journey, I can confidently say the difficulties faced during a PhD degree are part and parcel of becoming a good researcher. At this point, some engineering minds - like me - will deduce that the level of difficulty is directly proportional to the level of research quality. I would like to state that this is true (to serve as a morale booster), however my modified (researcher) brain thinks: no hard evidence exists to support this truth. One thing I can say for sure: the joy of performing research and creating something on your own is unmatched elsewhere in life. Personally, one of the best bits of being a PhD student is learning to become a problem solver, which is why I recommend to upcoming students to definitely take any chance you have to gain this experience.

First and foremost, I would like to thank Allah Almighty for his blessings in getting me this far. On a more earthly plane, I strongly believe that any research task should be taken in collaboration with other peers. I have been very lucky to have had a chance of learning from several people during my studies, and this thesis would not have been possible without the help and support of many such people, who have inspired me to pursue new directions, brought new insights to my work, and helped me to bring some reality to the absurd ideas that my mind brought forth.

I would like to thank Professor Jan Wikander who gave me the chance to begin my PhD studies, and who was always responsive to my ideas and research directions. I must also thank him for his critical remarks on my thesis. I would like to thank Micronic-MyData AB for supporting my research and Adjunct Professor Carl During for his co-supervision and for bringing an industrial perspective to my work. One of the first people I met at KTH was Professor Martin Törngren, who - seeing my interest in embedded systems - invited me to join the summer school that was then hosted by the mechatronics division. Martin’s enthusiasm, knowledge and expertise are big inspirations for me, although I must admit that I have not been very good at learning to juggle as well as he does. Another person I must thank is Professor Chris Paredis who invited me to Georgia Tech and took keen interest in my work. I have had many fruitful discussions with him, and his knowledge and insights in the subject are remarkable.
I would like to express my gratitude to Jonas Torry-Smith, Sebastian Herzig and Axel Reichwein for many interesting and productive discussions, and co-publications. I would also like to thank Professor Lucienne Blessing, Professor Mogens Myrop Andreasen and Professor Christian Weber, who gave me the opportunity to participate in the engineering design research summer school, and provided guidance on learning how best to perform research. I would also like to thank Professor Hans Vangheluwe, whose feedback on my Licentiate thesis lit up many light bulbs in my mind, and who is an exceptional person to have discussions with.

I would like to show my appreciation to my colleagues at the mechatronics division for an inspiring and dynamic research environment. I cannot forget lunch time discussions with Daniel M., Daniel F., Mohammad and Sagar. Thanks also to Magnus, Frederik L., Fredrik A., Martin G, Lei, Jad, Baha and Matthias for enlightening discussions and moral support. I am truly grateful to Vicki for proof reading this thesis and for providing motivational remarks on what I wrote.

Finally, I must thank my wife for her endless support and understanding, my parents, sisters and friends for believing in me, and for their encouragement throughout my work. I can perhaps see the light at the end of the tunnel now and I am excited for pursuing what lies ahead.

Ahsan Qamar
KTH, Stockholm, March 2013
# Contents

<table>
<thead>
<tr>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminology</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>ix</td>
</tr>
<tr>
<td>Contents</td>
<td>xi</td>
</tr>
<tr>
<td>List of Appended Publications</td>
<td>xv</td>
</tr>
<tr>
<td>List of Additional Publications</td>
<td>xvii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem formulation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Historical perspective on consistency management</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Scope of thesis</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Research questions and hypothesis</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Example problem</td>
<td>7</td>
</tr>
<tr>
<td>1.6.1 Situation 1: Assessing consequences</td>
<td>8</td>
</tr>
<tr>
<td>1.6.2 Situation 2: Lack of dependency management</td>
<td>10</td>
</tr>
<tr>
<td>1.7 Thesis overview</td>
<td>11</td>
</tr>
<tr>
<td>1.8 Chapter summary</td>
<td>12</td>
</tr>
<tr>
<td>2 Research Design</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Research methodology</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1 Descriptive study 2</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 Prescriptive study 2</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Descriptive study 3</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Research type</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Data collection methods</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Research evolution</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Chapter summary</td>
<td>21</td>
</tr>
<tr>
<td>3 State of the Art</td>
<td>23</td>
</tr>
</tbody>
</table>
### CONTENTS

3.1 Areas of relevance ........................................ 23
3.2 Mechatronic design ....................................... 24
   3.2.1 Challenges in mechatronic design .................. 25
3.3 Integration ................................................. 28
3.4 Dependency management ................................... 30
3.5 Consistency management .................................. 32
3.6 Conclusions on literature review ......................... 34
3.7 Chapter summary .......................................... 34

4 Dependency Modeling and Model Management ............ 37
   4.1 Decision-based engineering design ..................... 37
   4.2 Properties and the decision making process ........... 39
   4.3 Fundamentals of consistency management ............... 42
      4.3.1 Classification of consistency ...................... 44
      4.3.2 Consistency during model-based mechatronic design 45
   4.4 Fundamentals of dependency modeling .................. 48
      4.4.1 Properties ........................................ 49
      4.4.2 Dependencies ...................................... 51
      4.4.3 Dependencies across models (in different domains) 52
      4.4.4 Illustration ...................................... 54
      4.4.5 Modeling dependencies at different levels-of-detail 59
   4.5 The services based on dependency modeling .......... 64
      4.5.1 Executing dependencies ............................ 64
      4.5.2 Consistency checking .............................. 66
      4.5.3 Change management ................................ 66
      4.5.4 Human interaction ................................ 67
   4.6 The infrastructure for modeling dependencies ....... 68
   4.7 Modeling dependencies in SysML ....................... 71
      4.7.1 Dependency modeling between SysML and CAD ....... 74
      4.7.2 Illustration: dependency modeling process for a robot 76
      4.7.3 Dependency patterns ............................... 79
   4.8 Comparison to the Design Structure Matrix ........... 80
   4.9 The Dependency Modeling Language .................... 84
      4.9.1 DML Grammar .................................... 87
      4.9.2 Illustration: Dependency modeling through the DML 92
      4.9.3 Conclusions on the DML ......................... 93
   4.10 Chapter summary ....................................... 95

5 Discussion .................................................. 97
   5.1 Big picture ............................................. 97
   5.2 Abstractions between dependency models ............... 99
   5.3 Future work ............................................ 104
      5.3.1 Evolution of DML ................................ 105
      5.3.2 Role of an integration framework ................. 110
CONTENTS

5.3.3 Case study in an industrial setting ............................. 110
5.3.4 Relating Dependency Modeling and Design Contracts .......... 111
5.3.5 Design Process and Dependency Modeling ....................... 112
5.4 Chapter summary .............................................. 114

6 Conclusions 117
6.1 Relationships between appended papers ........................... 117
6.2 Communication between mechatronic domains ...................... 118
6.3 Dependency modeling ........................................... 119
   6.3.1 Consistency checking .................................. 121
   6.3.2 Assessing consequences .................................. 121
   6.3.3 Optimization ........................................ 122
6.4 Chapter summary .............................................. 123

7 Summary of Appended Papers 125
7.1 Paper A .................................................. 125
7.2 Paper B .................................................. 125
7.3 Paper C .................................................. 126
7.4 Paper D .................................................. 126
7.5 Paper E .................................................. 127
7.6 Paper F .................................................. 127

Bibliography 129

Appended Papers 143

A Paper A .................................................. 145
B Paper B .................................................. 157
C Paper C .................................................. 169
D Paper D .................................................. 193
E Paper E .................................................. 207
F Paper F .................................................. 225
List of Appended Publications

- **Paper A**

  Both Ahsan and Jonas are the first authors of this paper. Ahsan wrote Sections 3.3, 4, 5.2, and 5.3, and developed the tables. Jonas wrote Sections 3.1, 3.2, 3.4, and 5.1. Jonas also contributed with the information about the case study. The remaining sections of the paper were mutually developed. The remaining authors provided feedback for improvements.

- **Paper B**

  Ahsan and Chris Paredis developed the ideas behind the dependency modeling approach reported in this paper. The paper is mostly written by Ahsan, and Chris Paredis suggested some essential additions and improvements to the text in different parts of the paper. Jan and Carl provided paper review and suggestions for improvements.

- **Paper C**

  Ahsan developed the work reported in this paper, and wrote all the sections. Jan and Carl provided review and suggestions for improvements.

- **Paper D**
  Ahsan Qamar, Jan Wikander and Carl During, “Designing Mechatronic Sys-

Ahsan coordinated the bed case study with a group of 6 engineers, analyzed the information, and wrote the paper. Jan and Carl provided suggestions for improvement to the text.

- **Paper E**

  Ahsan performed the case study on robot design and wrote the paper. Martin provided suggestions for improvements of text in various parts of the paper. Jan and Carl provided a review to the paper.

- **Paper F**

  Ahsan developed the ideas and wrote the paper. Jan and Carl provide corrections and essential feedback.
List of Additional Publications


Chapter 1

Introduction

The development of mechatronic products is influenced by various factors. Complexity of the product itself, background of design teams, and the challenges of working across multiple domains are some of the major factors. This chapter discusses some of the main challenges in mechatronic design, and proposes solutions to overcome them.

1.1 Background

More and more products today are a combination of mechanical, electronic, and software components. This trend is evident in a wide variety of products such as home appliances, construction equipment, production systems, automobiles, and airplanes. In terms of engineering such products, the word *mechatronics* is used to differentiate them from purely mechanical products. In general, mechatronics represents products where certain capabilities are achieved through intelligent design and use of components from mechanics, electronics, and software domains - for instance, a capability to *control* the mechanical system. Mechatronics aims at systematically building up a harmonized product leveraging advantages from each of the involved disciplines.

One can ask the question: why should we develop mechatronic products? The answer lies in the fact that some systems today can only be realized as a mechatronic product - from a CD player or a digital watch, to a robot or a device to help your car’s, combustion engine meet emission legislation. In order to enhance the functionality and performance of traditional mechanical systems, engineers in the early days of mechatronics tried to utilize electronics and software to realize a number of additional product functions. This trend has increased to such an extent that a large number of previously mechanical functions are now at least partly realized through electronics and software. This facilitates modifying (enhancing) the behavior of a product by software modifications throughout the product life cycle, e.g., combustion engine control in an automobile through software modification of
the fuel injection system. Furthermore, systems which were previously developed independently of each other are now developed as a coupled system in order to provide new product features, e.g., brake and steer by wire systems. However, such couplings do make the systems more complex and product development more difficult to perform.

In agreement with Hansen and Andreasen (2004), it can be said that the decision to select a mechatronic alternative for a product (as compared to a purely mechanical alternative) is based on three dimensions. The first is attractiveness, i.e. performance properties in terms of user preferences; the second is economy, i.e. cost for the user; and the third is tractability, i.e. ability to proceed from design concept into a physically realizable design within allocated resources and time. In addition to these dimensions, a mechatronic product is also driven by dematerialization and compactness demands. The realization of functions in electronics and software rather than mechanics is a major driver for dematerialization, which aids in creating lighter and more compact products. Based on these demands, it can be said that through mechatronics, an engineering designer aims for the following benefits:

- Cost efficiency (performance related to cost)
- Dematerialization
- Flexibility to make changes throughout the product’s life phases (without changing the underlying physical systems)
- Eco-friendliness e.g., eco-cars
- Better performance, e.g. through advanced control functionality
- More functions inside a product

In order to attain the benefits of mechatronics, large resources are allocated to the development of electronics and software sub-systems. Braun and Lindemann (2007) explain that the additional costs in mechatronics as compared to mechanical products stem from: the augmented development activities (in different domains); cross-domain coordination; additional tests and integration processes; higher costs of spare parts in case of product failure; and new technologies for production and verification. In fact, many errors not spotted during the design process result in failure in the testing and integration phase, hence leading to an increased cost. To keep the cost under control, the challenges that lead towards complications in product development need to be well understood and addressed. Section 1.2 provides details of these challenges.

1.2 Problem formulation

In agreement with the VDI-guideline (Association of German Engineers, 2004), in this thesis the term mechatronic design is generally used when the mechanical en-
1.2. PROBLEM FORMULATION

eering, electrical engineering, and computer science disciplines interact during product development. The resulting product is referred to as a mechatronic product, and the term mechatronics is used as an umbrella term covering the whole subject area. The first investigation performed within this thesis is to find the challenges inherent in mechatronic design and development, both as represented in the current literature, and as identified through our own observations when involved in mechatronic design and development projects. Paper A considers the challenges alongside the current state of the art methods which aim to address them. This investigation helped highlight those challenges which still lack adequate design support, and formulate the associated problems. In the following, these problems and their root causes are described:

1. Nature of the product: A mechatronic product is multi-domain by nature. The product is complex in itself due to the large number of functions it performs (Braun and Lindemann, 2007; Tomiyama et al., 2007; Gausemeier et al., 2009a). Furthermore, while designing, each function could be realized as a complex combination of mechanics, electronics and software components (Gausemeier et al., 2001). This complexity of the product and of the integration between multi-domain components makes it difficult for design engineers to understand a mechatronic system as a whole. It also makes it difficult to assess the consequences of different design concepts in order to evaluate them (Tomiyama et al., 2007; Danilovic and Browning, 2007; Gausemeier et al., 2008; Buur, 1990; Salminen and Verho, 1989).

2. Nature of product design activities: During the process of designing mechatronic products, the overall design problem is partitioned into domain-specific problems managed by design teams consisting of domain experts. As a result, the product properties also spread across different domains. Many of these properties are dependent on each other (Tomiyama et al., 2007), meaning they affect each other - referred to as dependencies, as per Paper B. Typically, a large number of dependencies have to be managed during the design process, for instance between Mechanical Computer Aided Design (MCAD) and dynamic analysis; between MCAD and Finite Element Modeling (FEM); and between MCAD and system design.

In current practice in Model-Based Systems Engineering (MBSE), dependency management is mostly implicit. Experts carry the knowledge about dependencies in their minds, and the dependencies are not captured formally through a model. Due to lack of supporting methods for dependency management, design activities within domains are performed in isolation of each other. Most commonly, mechanical design is performed first, then - with the mechanical design frozen - electronic and software design is performed. Such lines of action require tedious efforts going back and forth between domains in order to manage the dependencies and reach a suitable solution. It is also
difficult to assess the consequences of different design choices for the involved domains.

It can be said that without a solution to manage dependencies, both the synthesis and analysis phases of the design process are more difficult to perform. In order to keep the cost of design activity under control, it is important to seek solutions addressing dependency management.

3. Background of engineers involved in the design activity: Engineers involved in designing mechatronic products are typically from different backgrounds, and are usually experts within one domain. It is difficult for them to understand all cross-domain design issues (Salminen and Verho, 1989; Danilovic and Browning, 2007; Buur, 1990). Furthermore, design methods, modeling languages, tools and terminology vary from one domain to another (Adamsson, 2004). The current mechatronic design process is governed by a wide variety of model-based design and analysis tools, catering to the demands and concerns of the involved stakeholders. In such diverse tool environments, communication between these stakeholders is a major concern, especially since designers need to understand not only each other’s stakes, but also the system as a whole (Frey et al., 2009). Again, this problem is influenced by the lack of dependency management solutions.

4. Nature of development organization for mechatronic products: Many organizations encourage design discussion involving experts from different domains. The idea of such meetings is to synthesize solutions utilizing inputs from domain-experts and to manage dependencies and improve communication across domain borders. However, the adapted solutions are mostly ad-hoc, and are not systematic enough. Typically, errors may arise due to dependencies not spotted during these discussions, or communication problems due to lack of known context and terminology.

1.3 Historical perspective on consistency management

In 1999, NASA lost the unmanned MARS Climate Orbiter (MCO). Engineers were preparing to celebrate the orbit entry of the MCO when the calamity happened. The orbiter was 100 km too close to the red planet than planned, causing massive overheating, and the 125 million dollar spacecraft was destroyed in an instant. Surprise followed when the investigators found the root cause of the problem to be a rather simple error (Stephenson et al., 1999). Two teams based at different locations together designed the navigation and propulsion system of the spacecraft, but one used English units and the other used Metric units. The resulting inconsistency caused an error in thrust calculations and the spacecraft’s destruction. Seven years later, inconsistent specifications between two design teams led to a 6.1 billion dollars loss for the Airbus A380 program (Wong, 2006). Catastrophic failures such as these
could be avoided if information consistency is not taken for granted during product development.

The question is: why is consistency management difficult? As per Finkelstein (2000), "in trivial cases such as the sky is blue, the sky is not blue the inconsistency is easy to spot". However, inconsistencies in a real design setting are much more challenging to detect. For instance, there could be assertions that are not semantically grounded, hence their precise meaning is unknown, or an inconsistency is distributed across many assertions (Finkelstein, 2000). The information itself may be changing rapidly and the dependencies which influence various entities not explicitly known. In the MCO’s case, the dependencies between analyses performed by both teams were not explicitly known. If they had been, consistency checks could have been implemented, which would have detected the inconsistent units of the property values. Similarly, with the Airbus case, the overall effect of dependencies was not considered, leading to an incorrect total length of certain cabling being predicted through the CAD software. In agreement with Finkelstein (2000) and Herzig et al. (2011), it can be said that detecting inconsistencies requires actions such as building semantically correct models and managing existing dependencies. Although through such actions we can make sure that more inconsistencies are avoided, it should be noted that consistency can never be fully ensured (Herzig et al., 2011).

### 1.4 Scope of thesis

Based on historical perspective and investigations into mechatronic design challenges, there is a clear need for research in dependency management, which in turn will support the building of consistency management solutions. Furthermore, support for communication and design integration between domains is currently error prone and further model-based methods need to be investigated to improve the situation. The work reported in this thesis focuses both on the dependency management problem and the problem of supporting communication and integration between domains. It is hoped that such efforts will make it easier for mechatronic designers to perform their work, so that they can make decisions more efficiently and effectively.

The initial research that led to the work reported in this thesis also focused on the challenge of a common language. Within Paper A, this challenge is reported as the “lack of common language to represent a concept”. The initial research reported in Qamar (2011) investigated proposed common languages within mechatronics, and studied in detail whether OMG SysML™ (Object Management Group, 2012) is a satisfactory common language to support mechatronic design. It was concluded that although SysML is sufficient as a common language, there could be a need for many domain-specific versions, for instance for automotive systems and for aerospace systems. This notion agrees with the idea of the Domain-Specific Languages (DSL) as discussed by Mosterman and Vangheluwe (2004). Another
part of the initial research was to investigate the gaps between the product synthesis phase, where informal sketches are used, and the creation of a common product model through the common language. It was concluded that there is a substantial gap between the synthesis phase and the creation of formal models. Modeling environments and encompassing technologies to support integration between freehand sketches and formal models are currently missing. However, research such as Macomber and Yang (2011) and Murugappan and Ramani (2009) are steps in the right direction.

1.5 Research questions and hypothesis

Based on the problem statement and the scope of this research, the following research questions are central to this thesis.

- **RQ1:** What kinds of dependencies exist between different domains in a mechatronic design context?
- **RQ2:** What type of approaches can be used to manage dependencies while performing design activities in different domains?
- **RQ3:** Given that there are dependencies between disparate models, how can consistency be ensured?
- **RQ4:** How can we support communication e.g. in terms of information exchange between design domains in a mechatronic design context?
- **RQ5:** Given that there are dependencies between disparate models, what is a suitable method for performing design optimization?
- **RQ6:** During design a number of alternative concepts may be considered. How can the consequences of choosing between these alternatives be assessed?

The following hypotheses aim to guide research towards answering the above questions:

- **Hyp 1:** By investigating dependencies between design domains through different case studies, it is possible to find out the nature of dependencies and the requirements to manage them (addresses RQ1).
- **Hyp 2:** An approach that makes dependencies explicit by capturing them through a formal model provides a foundation to manage dependencies (addresses RQ2).
- **Hyp 3:** Making dependencies explicit in a dependency model provides the necessary information to facilitate consistency management (addresses RQ3).
• **Hyp4:** To facilitate information exchange between disparate tools, a model-based strategy supported by tool adapters and model transformations helps in providing the communication support (addresses RQ4).

• **Hyp5:** Dependency models provide adequate details in order to support a design optimization where disparate models are lumped into an overall design study (addresses RQ5).

• **Hyp6:** Dependency modeling supports assessment of consequences for alternative concepts (addresses RQ6).

Based on the solutions stated in the above hypotheses, some further refinements are proposed as follows:

• Refinement for RQ2:
  – **RQ7:** How detailed should the dependency representation be?
  – **RQ8:** How can dependency models be managed, and how will humans utilize these models?
  – **RQ9:** What type of support is available in current modeling languages to model dependencies explicitly?
  – **RQ10:** How can the value gained be quantitatively compared against the effort required in creating and managing dependency models?

• Refinement for Hyp4:
  – **Hyp7:** A tool integration framework helps in creating tool adapters in a systematic fashion, and provides a sustainable infrastructure to support communication between disparate tools (addresses RQ4).

During the course of this thesis, we will relate our proposals and arguments to the above research questions and hypotheses. In Chapter 6, we will revisit the research questions and value the thesis work against them.

### 1.6 Example problem

In order to better reflect on the problem statement and the aimed intended support, this section will provide a visual representation of the problem for a simple mechatronic system. The objective is to gain a better understanding of the nature of the problem, and explain how we intend to provide solutions to it. The section is inspired by the visualization exercise proposed by Mogens Myrup Andreasen as explained in (Blessing and Chakrabarti, 2009). A fairly simple two degrees-of-freedom robot is chosen as an example system as shown in Figure 1.1. This robot consists of two mechanical arms actuated through DC motors, with sensors measuring the amount of rotation of the joints. The purpose of the robot is to move an object
Figure 1.1: Example mechatronic system: a two degree-of-freedom robot.

from a pickup point and drop it at the place point in the presence of an obstacle. The following section presents different situations where problems are faced while designing the robot.

1.6.1 Situation 1: Difficulty in assessing consequences of alternative design concepts

Electronics used to drive the motors are packaged as a hardware unit. It is not clear where to best place the unit or which technology to use for the communication. Several solutions are possible, spreading across different mechatronic domains. The unit can be placed on one of the arms (Figure 1.2(a)), which would then affect the load balancing on the mechanical structure but has the advantages of minimizing the wiring and simplifying the cooling of electronics (by using the mechanical structure as a cooling element). It also influences the reliability of the product. Another solution is to place the unit away from the robot, which adds extra cost for the wiring and extra components for cooling (Figure 1.2(b)). A wireless solution can be undertaken as shown in (Figure 1.2(c)), adding the cost of wireless transceivers, but providing advantages in terms of placing the unit away from the robot, which gives better protection, easier maintenance, and could provide further benefits if the robot is used in a closed environment. However, a wireless solution introduces constraints on the robot controller, which was originally designed to control over a wired network. Ultimately, it is not clear whether it is more important to consider
Figure 1.2: Solutions proposed for the hardware unit of the robot. (a) hardware unit mounted on the robot arm, (b) hardware unit placed away from the robot with wired connection (c) same as (b) but with wireless connection.

The situation above reflects the most common influence of dependencies on the design process. Each alternative solution, when modeled, has properties spread across different domains. The dependencies between these properties need to be managed in order to find the net utility of each alternative. This requires considering the overall effect of several dependencies, a process much easier to perform once the dependencies are captured explicitly through a dependency model.
1.6.2 Situation 2: Lack of dependency management

The mechanical, electronic and software development of the robot is supported through domain-specific development activities, and there are dependencies between properties captured through models in different domains. For instance, the workspace of the robot is determined by the physical properties; however, the physical properties also affect the controller design. The design of software is furthermore dependent on the processing power of the hardware. The performance of the robot has influences from all domains, e.g. related to motors, sensors, mechanical properties, hardware and software design. The problem here is again managing dependencies between disparate models, and we propose that an explicit dependency model is required to tackle this.

Figure 1.3(a) visualizes a scenario where the property controller performance of the robot is affected by the models of mechanics, dynamics, electronics, and software. By following the dependency modeling process, information relating to the property controller performance is extracted from each model. This information can then be used to represent the dependency relationships inside a dependency model. For instance, the mechanical properties of the robot (length and inertia) affect controller design and hence controller performance. Figure 1.3(b) visualizes our intended solution where a dedicated tool is used to create a dependency model, where dependencies to the property controller performance are represented. The dependency modeling tool is supported by an integration infrastructure which supports information transfer between disparate tools.
1.7 Thesis overview

This thesis is divided into seven chapters based on a set of six papers. In the following, a brief overview of each chapter is provided.

Chapter 1 provides an introduction to the thesis, and discusses the problems that are attacked within this research. The mechatronic design challenges faced are listed in order to frame the problem statement. The main research questions and hypotheses of this research are also established in this chapter.

Chapter 2 explains the research design, which is largely based on the design research methodology (DRM) proposed in Blessing and Chakrabarti (2009). The chapter presents details about the research clarification, descriptive and prescriptive studies, with an explanation of the methods used to develop these studies. Their contributions are highlighted in relation to the research questions crafted in Section 1.5. The chapter also classifies the thesis work in terms of the known design research types (Blessing and Chakrabarti, 2009). This helps in clarifying what different stakeholders can expect after the completion of the research project this thesis is based on.

Chapter 3 provides a short explanation on descriptive studies 1 and 2, where the aim is to gain understanding of the current situation through a literature search performed in relation to areas of relevance for the thesis topic. Through the literature review, this chapter highlights the important challenges relevant to research in mechatronic design, and identifies what kind of support is currently available for those challenges. This facilitates in completing the reference model of the research study, and determining the key factors that might be suitable to address in the prescriptive stage. Then, an impact model (as per DRM) is also constructed to identify the influence of the intended support.

Chapter 4 provides details about the second prescriptive study (with prescriptive study 1 reported in earlier work in Qamar (2011)), where the focus is towards development of an adequate method to manage dependencies. This chapter aims to form a strong theoretical foundation for the thesis work. The Dependency Modeling Language (DML) that came out as a result of the work is presented in this chapter, and the value of a dependency model built through a dedicated DML is illustrated. Through an integration case study between a CAD model and a SysML model, the chapter also illustrates the use of model transformations to support integration and communication between models.

Chapter 5 presents a discussion based on further questions arising as a result of architecting the DML. The impact model is revised here and how to evaluate the developed support against the success criteria is discussed. This chapter also highlights future work based on evolving the DML to support a list of intended features, formal testing of the DML semantics, and the use of integration frameworks. Further work targeting DML use in an industrial case is also discussed. The impact model is revised to show the potential influence of future work.

Chapter 6 concludes the thesis by summarizing the findings gained in the context of the proposed research questions, and aims to highlight the scientific contributions
of the performed work. Since this thesis is based on a number of appended papers, a brief summary of each publication is provided in Chapter 7.

1.8 Chapter summary

In this chapter, we have put together a picture of mechatronic product design, and discussed the factors that contribute to the challenges currently faced by mechatronic designers. Based on the challenges, the chapter provides a brief introduction to the goals and scope of this research, helping to formulate the main research questions for this thesis. A simple visualization of the problem statement was presented to the reader in order to provide a practical understanding of the challenges, and the support envisaged as helpful in addressing them.
Chapter 2

Research Design

The scientific aim of this research is to understand the challenges designers face while designing mechatronic products, and provide support to address some of the key challenges. These aims are expressed in the form of the research questions listed in Section 1.5. This chapter explains the research methodology adopted to answer the formulated research questions and to verify or nullify the chosen hypotheses.

2.1 Research methodology

The overall plan for the research project, of which this thesis is a part, was devised using the Design Research Methodology (DRM) by Blessing and Chakrabarti (2009) due to the project’s emphasis on engineering design. DRM provides a step-wise approach through which design research can be best undertaken. It is based on different phases, each devoted to a particular aim, starting with research clarification phase, where the aim is to define the main research problem, and continuing with the descriptive study, which is devoted to understanding the current situation with reference to the research problem. After identifying the factors that have led to the current situation, the areas in which support should be developed can be determined. Based on the research questions listed in Section 1.5, an overall research plan following the DRM can be formulated as shown in Figure 2.1. Note that the overall research plan is divided into three parts: the first includes the earlier work reported in (Qamar, 2011); the second covers the work reported in this thesis, and the third focuses on future work. The studies performed in relation to the research questions for earlier work are already discussed in (Qamar, 2011). In this section, we will focus on the studies performed to answer the research questions reported in Section 1.5.

The work reported in this thesis begins with descriptive study 2, where the current situation in dependency modeling is assessed, and followed by prescriptive study 2, where a Dependency Modeling Language (DML) is developed. Descriptive study 3 includes planned future work, focusing on further development and eval-
uation of the DML. The research approach illustrated in Figure 2.2 is followed to find answers to research questions 1-10. A short description of each stage of the research approach will be provided. Note that Figure 2.2 includes earlier work too, much of which is not discussed within this thesis but is partially covered through appended papers A, D, E, and F.

2.1.1 Descriptive study 2

The fifth stage of the overall research approach is focused on analyzing the nature of dependencies within mechatronic design and systems engineering through the available descriptive and prescriptive literature. Some of the descriptive literature such as Torry-Smith and Mortensen (2011), Tomiyama et al. (2007) and Buur (1990) hints towards the role of dependencies in mechatronic design, but does not provide an explicit consideration towards modeling dependencies.

To gain an in-depth understanding about how dependencies come into play during the design process, a case study on a fairly simple two degree-of-freedom robot was performed. By modeling dependencies manually for the robot example, the nature of dependencies could be investigated. This provided substantial information not only about the nature of dependencies (stage 6), but also about what features are required in a modeling language that will support dependency modeling. Key considerations included how detailed the dependency representation should be, and how to manage the dependency models - which provided some answers to RQ1, RQ2, RQ7 and RQ8. Modeling dependencies explicitly offers a number of benefits - such as managing an engineering change order and maintaining consistency - although whether these benefits outweigh the cost of modeling remains unclear at this stage. Therefore, it was decided that further work is required in order to quantify the value gained against the effort required to model and manage dependencies. Furthermore, the human interaction with the dependency network is another area which needs exploration, providing some guidelines for handling RQ8 and RQ10, although more work is needed in order to answer them.

From the prescriptive literature on dependency modeling, the Design Structure Matrix (Eppinger and Browning, 2012), PIDO tools such as ModelCenter (Phoenix Integration, 2012), and dependency modeling through SysML were considered. The ideas about dependency modeling (developed though the robot example) were compared against these approaches, firstly considering what support is available in current modeling languages. Since SysML is one of the languages currently available that can support dependency modeling, a case study between SysML and CAD was undertaken, where dependencies between system hierarchy and CAD assembly were represented in a SysML model. This provided answers to RQ9. The DSM and PIDO tools were also considered against the dependency modeling approach, providing further answers to RQ9.

Through the studies performed on dependency modeling for a robot, dependency modeling in SysML, and comparison to other state of the art approaches, a proposal
2.1. RESEARCH METHODOLOGY

Figure 2.1: The overall research plan, including earlier work (Qamar, 2011), this thesis, and future work.
CHAPTER 2. RESEARCH DESIGN

Figure 2.2: The research approach of the project of which this thesis is a part.
2.1. RESEARCH METHODOLOGY

for a set of features required in the DML is put forward (stage 7). In prescriptive study 2, the intended support will be developed based on these features.

2.1.2 Prescriptive study 2

Prescriptive study 2 corresponds to the eighth stage in the research approach, and is focused on development of the intended support, i.e., development of the Dependency Modeler, a tool encompassing the DML where dependency models can be built and analyzed. Here, the Dependency Modeler was supported through the Eclipse Modeling Framework (Eclipse Foundation, 2009a), which provides common ground to represent disparate models. Within the time frame of this research project, it was not possible to implement all the required features of the DML, thus the focus was on those that represent the core foundations, without which the DML would cease to exist. Through these features, the Dependency Modeler enabled modeling dependencies for the robot example used previously in descriptive study 2. The resulting dependency network was studied to find the potential benefits for the designer. These investigations provided the initial proof about the functionality of the developed support. Some of the answers to RQ3 about managing consistency were available through the robot dependency model. However the resulting implementation of a consistency checker on top of the dependency model was not considered in the work reported here. Therefore, although the answers to RQ3 were partially available, hypothesis 3 (addressing RQ3) was not fully verified. Future work may include implementing a consistency checker in order to provide this proof.

Study of a scenario for design optimization found that it leads to dependency loops - containing explicit information of the involved dependencies - along with a distinction between the optimization reference the optimization result, which brought an answer to RQ5.

Another scenario considered was working with alternative design solutions, and it was found that this leads to multiple dependency networks, hence providing some answers for RQ6. To answer RQ6 completely requires a design study where alternative solutions are evaluated based on their dependency networks; such a study could be seen as a possible future work to this thesis.

Model transformations were implemented between a SysML tool and a CAD tool to test the potential for supporting communication and integration between models and tools. Corresponding tool adapters were also necessary in order to automate such transformations. Although found to be technically possible, building tool adapters and transformations for point-to-point solutions such as SysML-CAD is neither scalable, nor efficient. Therefore, although some answers were found for RQ4, it was considered that future work on solving this problem should focus on integration frameworks, since they provide guiding principles, interface specifications, and technology base to perform tool integration in a systematic fashion.
2.1.3 Descriptive study 3

These studies (below), detailed in Section 5.3, are included as suggested future work, mainly targeting the following:

- Evaluate dependency modeling through DML supported by an integration framework.
- Evolve the DML with a complete set of features.
- Implement consistency checking for the DML.
- Develop and evaluate the visualization of dependency network for human interaction.
- Perform verification of semantics of the DML through formal methods.
- Test the dependency modeling approach in terms of effort/cost required against the benefits gained through an industrial case study.

2.2 Research type

This research project centers around a review-based research clarification phase. As per the DRM, the earlier work part of the project was reported as research type 6 (Qamar, 2011). The work reported in this thesis centers around a review-based descriptive phase, since only current literature and experiences from mechatronic design case studies are used to reach the conclusion that dependency modeling is essential, and no empirical studies were performed in this phase. The prescriptive phase is a comprehensive one where the Dependency Modeler supporting the DML is developed. This is followed by an initial evaluation in the third descriptive phase. Based on this, the research reported in this thesis is classified as research type 3 according to DRM, as shown in Figure 2.3.

2.3 Data collection methods

Different data collection methods have been utilized to address the formulated research questions and the chosen hypotheses. Figure 2.4 shows the question-method matrix as proposed in (Blessing and Chakrabarti, 2009, Chap.4). The figure shows which method is utilized to address a particular research question and hypothesis. It also shows whether the answer to a research question or the verification/nullification of a hypothesis can be fully (‘xx’) or partially (‘x’) obtained through a particular method. For example, descriptive and prescriptive literature on dependency modeling along with a case study on modeling dependencies for a robot helps in finding the nature of dependencies (RQ1). Hypothesis 2 is fully supported not only by the case study but also by the integration framework case study. For selection purposes, the effort required to perform each case study can provide a
### 2.4. RESEARCH EVOLUTION

Figure 2.2 illustrates the evolution of the research to date. It is interesting to note that the need for dependency modeling was not apparent at the start of the research. In fact, until stage 4 (in the earlier work), explicit dependency modeling

<table>
<thead>
<tr>
<th>Earlier Work</th>
<th>Research Clarification</th>
<th>Descriptive Study 1</th>
<th>Prescriptive Study 1</th>
<th>Descriptive Study 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Review-based</td>
<td>Comprehensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Review-based</td>
<td>Comprehensive</td>
<td>Initial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Review-based</td>
<td>Review-based</td>
<td>Comprehensive</td>
<td>Initial</td>
<td></td>
</tr>
<tr>
<td>4. Review-based</td>
<td>Review-based</td>
<td>Comprehensive</td>
<td>Initial/</td>
<td></td>
</tr>
<tr>
<td>5. Review-based</td>
<td>Comprehensive</td>
<td>Comprehensive</td>
<td>Initial</td>
<td></td>
</tr>
<tr>
<td>6. Review-based</td>
<td>Review-based</td>
<td>Comprehensive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Review-based</td>
<td>Comprehensive</td>
<td>Comprehensive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>This Thesis</th>
<th>Research Clarification</th>
<th>Descriptive Study 2</th>
<th>Prescriptive Study 2</th>
<th>Descriptive Study 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Review-based</td>
<td>Comprehensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Review-based</td>
<td>Comprehensive</td>
<td>Initial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Review-based</td>
<td>Review-based</td>
<td>Comprehensive</td>
<td>Initial</td>
<td></td>
</tr>
<tr>
<td>4. Review-based</td>
<td>Review-based</td>
<td>Comprehensive</td>
<td>Initial/</td>
<td></td>
</tr>
<tr>
<td>5. Review-based</td>
<td>Comprehensive</td>
<td>Comprehensive</td>
<td>Initial</td>
<td></td>
</tr>
<tr>
<td>6. Review-based</td>
<td>Review-based</td>
<td>Comprehensive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Review-based</td>
<td>Comprehensive</td>
<td>Comprehensive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.3: Design research type, figure adapted from Blessing and Chakrabarti (2009).

measure of benefit of one over the other and additional information is therefore included underneath each box in Figure 2.4, where high/low effort for the researcher (‘RR/R’) and the participant (‘PP/P’) is represented. Note that Figure 2.4 also includes methods which are proposed as future work, for instance case study using an integration framework.
### Chapter 2. Research Design

**Analyze literature**

**Case study on dependency modeling**

**Case study on use of model transformations**

**Case study using DML**

**Case study using integration framework**

**Assessment exercise**

| RQ1 | **X** | **X** |  |  |  |  |
|-----|-------|-------|  |  |  |  |
| RQ2 | **X** | **X** |  |  |  |  |
| RQ3 | **X** | **X** | **X** | **XX** | **XX** |  |
| RQ4 | **X** |  | **XX** | **XX** |  |  |
| RQ5 | **X** | **X** |  |  | **X** | **X** |
| RQ6 | **X** |  | **X** | **X** |  | **XX** |
| Hyp1 | **X** |  |  |  |  |  |
| Hyp2 | **X** |  | **XX** | **XX** |  |  |
| Hyp3 | **X** |  | **X** | **XX** | **XX** |  |
| Hyp4 | **X** |  | **XX** | **XX** |  |  |
| Hyp5 | **X** | **X** |  | **XX** | **XX** |  |
| Hyp6 | **X** | **X** |  | **X** | **XX** |  |

| Hyp1 | **X** |  |  |  |  |  |
| Hyp2 | **X** |  | **XX** | **XX** |  |  |
| Hyp3 | **X** |  | **X** | **XX** | **XX** |  |
| Hyp4 | **X** |  | **XX** | **XX** |  |  |
| Hyp5 | **X** | **X** |  | **XX** | **XX** |  |
| Hyp6 | **X** | **X** |  | **X** | **XX** |  |

**Figure 2.4: Question-method matrix showing the data collection methods used to answer the research questions.**
was not even considered. The earlier work focused on evaluating a common language supporting conceptual design of mechatronics, along with the belief that the SysML model would serve as a common product model among disparate models to support integration and communication. The main idea behind this approach was that through SysML profiles and model transformations, each domain-specific model can be integrated with the SysML model, and this integration could be used for other services such as dependency management.

In this thesis, we deviate from this approach, instead identifying dependency management as the main challenge, with sound methods for dependency capture conspicuously lacking (stage 5). It was at this point that the dependency modeling approach was formulated as an alternative to the approach of using SysML as a central product model for product data integration. Through modeling dependencies, point-to-point integration between models may no longer be required. Instead, dependencies will be captured and managed explicitly. For further details, the reader is referred to Chapter 4 of this thesis.

2.5 Chapter summary

This chapter provided details about the evolution of the overall research plan, from the earlier work, through work reported in this thesis, and onto planned future work. Based on the DRM, the research approach was broken into different stages including descriptive and prescriptive studies. The chapter also discusses the data collection methods used to find answers to the formulated research questions and hypotheses.
Chapter 3

State of the Art

This chapter aims to describe the literature relevant to the research project of which this thesis is a part. The chapter provides details on descriptive study 1, carried out in earlier-work, and on descriptive study 2, which is a part of this thesis. The overall aim, through both studies, is to understand the current problems within the design of mechatronic products, and identify the factors that have led to this situation. Both studies are review-based, searching relevant literature to find evidence of a problem, or proposed solutions to tackle a particular problem. The gaps that this research aims to fill are highlighted through the state of the art description.

3.1 Areas of relevance

Figure 3.1 shows the Areas of Relevance and Contribution diagram (ARC) for this project, including the central research topic, i.e. *model and dependency management in mechatronic design*. The areas in which the scientific contributions are made (*areas of contribution*) are highlighted. In accordance with Blessing and Chakrabarti (2009, Chap. 3), while looking for relevant literature, it is important to look also at disciplines other than one’s own field of research for interesting theories, studies, methods and concepts. Therefore a number of potentially relevant areas - not directly related to the research topic of this thesis - were looked at, including mathematics, psychology and design space exploration. These are marked as *useful areas* in Figure 3.1, whilst other areas directly related to the topic of this thesis are marked as *essential areas*.

The overall research topic can be divided into three main areas: engineering design, mechatronics and integration. Design research in itself contains many design theories and approaches. Mechatronic products are multi-domain by nature, hence research in mechatronic design spans many areas. For instance, design modeling is an important part of engineering design; design modeling for mechatronics spans across different domains; and each domain in turn employs different design methods, modeling languages and tools. Thus the third area, integration between domains
Dependencies between design activities and the corresponding artifacts currently require vital but time consuming attention from the designers. They influence not only the design concepts that come out of different domains, but also the design costs. In addition, managing dependencies is closely connected to managing consistency between disparate models. Therefore, integration will also be looked at from the perspective of dependency management and consistency management between design models. In the following sections, we briefly overview the state of the art for mechatronic design, integration, consistency management and dependency management.

3.2 Mechatronic design

To find the relevant contributions within mechatronic design, a comprehensive literature study was carried out, as reported in Appended Paper A. The study covers relevant conferences within IEEE (The Institute of Electrical and Electronic Engineers, 2013b), ASME (American Society of Mechanical Engineers, 2013), De-
sign Society (The Design Society, 2013), and INCOSE (International Council on Systems Engineering, 2013). The selected journals searched for relevant articles included: Journal of Mechanical Design, Research in Engineering Design, Systems Engineering, CIRP Annals - Manufacturing Engineering, Elsevier Mechatronics, IEEE/ASME Transactions on Mechatronics, and Journal of Engineering Design. The range of conferences and journals in this study does not make the search exhaustive, however a deeper review of the selection and their references aims to offset any low-level author bias that may be present. In addition, a number of contributions from relevant mechatronics research groups around the world were read, and shortlisted based on their relevance to mechatronic design.

3.2.1 Challenges in mechatronic design

The first step performed in the literature search was to compile a list of statements regarding challenges in mechatronic design. The list of challenges was studied further through a case study to figure out whether certain challenges remain undiscovered in the available literature. As detailed in Paper A, the most commonly reported sets of challenges are primarily related to the way a product concept can be described and how information linked to a product concept can be shared across different engineering disciplines.

The challenges identified in Paper A that are still not addressed in the current literature were identified as:

- Difficulty in assessing consequences of selecting between alternative concepts.
- Lack of a common language to represent a concept.
- The transfer of models and information between domains.

**Difficulty in assessing consequences of selecting between alternative concepts**

Designers are frequently faced with alternative concepts that need to be evaluated against each other. One such situation is explained in Paper A, where a solution to a power consumption problem leads to three alternatives (redesign of electronics, software improvement, or battery change/life phase), where it is difficult to weigh one over the other due to insufficient knowledge of the effects and the required design effort. Clearly, one of the main challenges is that there are often different ways of solving a design problem, and the subjective reasoning involved in selecting one alternative over another becomes even more difficult for mechatronics due to the designers’ potential lack of awareness of the cross-disciplinary impact.

The reviewed literature proposes several ways to help assess the consequences of a design concept. One such solution is relationship management techniques like the Design Structure Matrix (DSM)/Domain Mapping Matrix (DMM) (Eppinger and Browning, 2012), QFD (Hauser and Clausing, 1988), and FunKey Architecting
(Bonnema, 2008). These approaches aid in modeling relationships between the functions, and components of a product, along with taking user preferences into account. However, despite being useful to gain understanding of multiple relations across domains, it becomes too cumbersome to analyze the consequences of a design choice through them, due to the effort required and efficacy of the method (Paper A, Paper B).

Other approaches that can be useful for the assessment of consequences include informal description of a design concept such as A3 overviews (Borches and Bonnema, 2010b), semi-formal specification language (SFSL) for mechatronics (Gausemeier et al., 2001), and using formal modeling such as SysML. A3 overviews do not support mechatronic specific aspects, such as the implication of the different allocation of functions within domains, as explained by Welp and Jansen (2004). On the other hand, more formal approaches like SFSL and SysML are good to model different design concepts, and possibly to also gain understanding about their properties through integration with simulation and analysis languages such as Modelica (Modelica Association, 2010). However, to assess an alternative concept, models which capture different properties of each alternative need to be grouped into an overall design study, meaning that dependencies between models need to be managed, although this is not easy due to only implicit information (about dependencies) being available to the designers (Paper B).

**Lack of a common language to represent a concept**

Mechatronic design is usually performed by a multi-disciplinary team of domain experts, who are unlikely to possess inter-disciplinary knowledge to get a detailed enough understanding of the whole design problem. Therefore, it is difficult to establish a common mechatronic view - instead, different domain-specific views are established and the dependencies in between are unclear. To establish a common understanding, especially during the conceptual phase of the development, requires a medium that permits designers to effectively communicate with each other (Paper A). This has been regarded as a common language issue by many researchers within the mechatronic community such as in (Tomiyama et al., 2007; Woestenenk et al., 2010; Gausemeier et al., 2001; Adamsson, 2004; Buur, 1990; Salminen and Verho, 1989).

One of the proposed solutions for a common language is to utilize methods based on functional thinking, and many design approaches exist based on this, including:

- Functional Design Framework (FDF) (Nagel et al., 2008).
• Axiomatic design by Suh (2001).

Erden et al. (2008) provides a good review of different functional modeling approaches and their applications. However, functional thinking is only a part of the picture of design activities, and it fails to support other factors such as structural aspects and how component choice and configuration affects system properties (Paper A). Functional modeling is abstract in terms of the level at which the description of the product concept is performed. Therefore, it can only serve the purpose of being a common language to an abstract level. Furthermore, it is typical that after the functional modeling the development process becomes domain-specific.

Managing design activities through requirements is also proposed as a possible solution to the common language issue. Examples include the Systems Engineering process (Sage and Rouse, 2009), and the work performed by Woestenenk et al. (2010). However, although they can be used for goal specifications (of the product to be), or result specifications (documenting the final product), requirements are largely used to control a design concept rather than represent one, which is the sole purpose of the common language discussed here.

A holistic approach towards modeling based on informal and formal representations is proposed by different groups of researchers as a solution to the common language issue. One example of informal description is the A3 architecture overviews. This approach aims to provide an overview of the complete system architecture in terms of different system aspects, such as functional and physical aspects. All these aspects are represented on an A3 size paper, providing limited but useful information needed by the designer not only to represent their design concept, but also to be able to communicate with other domain experts during a multi-disciplinary design process (Borches, 2011). As an approach, A3 overviews has the same potential of becoming a common language as functional thinking, but they carry the same drawbacks. The complexity in mechatronic design is not manageable only through such approaches.

In line with the holistic design approach, the Semi-Formal Specification Language (SFSL) (Gausemeier et al., 2001, 2007, 2010, 2009b) is a language more specifically related to mechatronics. It specifies a mechatronic concept in terms of a number of aspects, such as function and active structure. Proponents of SFSL argue that a mechatronic system needs to be described through different views, thus leading to different aspects inside SFSL. However, we believe that such a representation will also lead to domain-specific concretization during the initial design phases. SFSL proposes different semantics related to each mechatronic aspect, with the aim that engineers with different background can relate themselves to different aspects supported by SFSL. In that sense, it is not a pure common language, but can be treated as a language targeting different mechatronic aspects during the conceptual design phase.

A different approach to support Model-Based System Engineering (MBSE) is SysML, where a system is represented completely in terms of its structure and behavior through different SysML diagrams (Friedenthal et al., 2008, Chap. 3). This
helps in establishing a holistic approach towards system design based on a formal language. Wölkl and Shea (2009) and Follmer et al. (2010) explain the usefulness of SysML in the conceptual design phase of product development. This approach is somewhat similar to the SFSL, although (Gausemeier et al., 2009b) suggests that SysML is not suitable for representing a mechatronic concept during the conceptual phase since it does not cover different mechatronic aspects, hence leading to the creation of SFSL. This implies that SysML and SFSL differ in the way aspects are treated. On the other hand, Borches and Bonnema (2010a) document a conclusion that contradicts both SFSL and SysML by showing the results from an empirical research performed in Philips. The authors show that formal modeling approaches such as SysML do not usually solve the communication problems between people with different backgrounds, nor do they produce models that are intuitive and easy to understand. It can be concluded here that approaches such as SysML, SFSL and A3 overviews can all prove to be beneficial in eliminating a reasonable subset of the problems currently faced in conceptual design of mechatronics. However, based on the product domain, different common languages may still be required.

Transfer of models and information between domains

This challenge is similar to the topic of integration discussed in Section 3.3 (below).

3.3 Integration

The third challenge reported in Paper A as not sufficiently addressed by current support is the transfer of models and information between domains. One of the main factors here is the lack of dependency management currently prevalent in mechatronic design.

The multi-disciplinary teams involved in designing mechatronic products carry out domain-specific development in order to move forward in the design process, leading to the creation of different domain-specific design models, or domain specific views. It is necessary to establish some means of communication between such views in order to avoid integration problems later. Frey et al. (2009) classifies the communication probabilities between two design domains in terms of: person-person, method-method, model-model, and analysis tool-analysis tool. A person-person communication is regarded as error prone, and method-method communication is not always possible due to lack of analogy between two completely different methods within different domains. Analysis tool-analysis tool communication is possible, although it is based on execution (e.g. co-simulation) of developed models and not their development. A model-model communication between two domains is possible, provided that the required information is available explicitly and correctly through models.

The communication problem might not be that big, provided a broadly accepted methodology for mechatronic design gains acceptance. The VDI2206 guideline (Association of German Engineers, 2004) has been proposed, but does not discuss how
to manage the dependencies between mechatronic domains, nor does it consider how to address the problem of abstraction between design models and their evolution during design iterations. This further explains the need for model-model communication approaches.

One possible solution to support communication is through data exchange standards. For instance, within the CAD community a number of international standards have been developed to make product data exchange possible. Among these standards, the ISO10303 or STEP standard (Pratt, 2001) is probably one of the most well-known. However, as discussed by Gielingh (2008), the industrial uptake of STEP and other popularized standards has been poor. Some key issues, such as errors in exchanged data and loss of data when using a standardized neutral model, are still not resolved (Gielingh, 2008). Moreover, tool-specific implementation of standards may not strictly conform, therefore exchanging data between heterogeneous tools may lead to errors. As a result, organizations tend to rely on tools from a single vendor (which can be costly), or stick to documents and meetings for data exchange (which can be time consuming).

In parallel to the STEP standard, the WEB Ontology Language (OWL) (W3C OWL Working Group, 2002) has been proposed for defining vocabularies, essentially capturing knowledge about a domain. However, this translates to many ontologies covering the conceptual space of different domains. Again, it is difficult to have a single ontology covering the conceptual space of the whole product and its life cycle (Gielingh, 2008).

Use of the model transformation technology to integrate disparate models has been reported in several publications. Engelson et al. (2003) showed the use of domain-specific Modelica libraries to perform multi-domain dynamic analysis and controller design tasks on a mechanical system represented by CAD models. Bhat-tacharya et al. (2006) provided an example of integration between CATIA (Dassault Systems, 2011) and Modelica. Chen and Schaefer (2007) discussed integration between Electrical-CAD (ECAD) and Mechanical-CAD (MCAD) models. Many of the approaches are targeted towards integrating a system model built through languages such as SysML or UML with a corresponding analysis model. This potentially means raising the level of abstraction by going from a domain-specific model towards a system model. In some cases, the system model is also used for establishing relationships between domain-specific models. Examples include:

- Integration of UML and Simulink (Hooman et al., 2004; Sjöstedt et al., 2008; Vanderperren and Dehaene, 2006; Brisolara et al., 2007; Boldt, 2006).
- Integration of SysML/UML and Modelica (Johnson et al., 2012; Schamai et al., 2011, 2010, 2009).
- Integration of SysML and ECAD (Shah et al., 2009).

Cao et al. (2011) discussed SysML extensions to support modeling of discrete/continuous hybrid dynamic behavior of mechatronic systems. Shah et al. (2010)
explained a multi-view modeling approach based on the integration of a structural view in SysML with an electrical design (ECAD) view and with the analysis view in Modelica. Kerzhner and Paredis (2011) formulated a general framework for capturing a design problem in SysML, and for relating it to analysis and test cases through graph transformations.

Although integrating disparate models in the above fashion is a solution to information exchange, the dependency management is still inadequate. Model transformations can support modeling and management of dependencies only if they are constructed to support dependency modeling - as per the proposals of this thesis.

A systematic way of establishing communication between tools is by utilizing an integration framework. For instance, iFEST (Törngren et al., 2012a) is a proposal encompassing an integration framework (principles, interface specifications, guidelines and a integration technology base) as well as supporting tools, framework-compliant platforms and adaptors for commonly used embedded systems tools. The idea is that design activities are distributed over a diverse tool set, and each tool is considered as both provider and consumer of services and data to/from other tools. There is no central platform between tools, rather tools communicate directly with each other at run-time. Note that this approach is NOT to be confused with ad-hoc, point-to-point integration of tools, since each tool’s services and data are defined according to a pre-determined specification, allowing this same interface to be deployed in any other tool chain configuration. In addition to an integration framework, tool chains can be built utilizing a language that supports building tool integration solutions. The Tool Integration Language (TIL) (Biehl et al., 2012) and Cyphy (Simko et al., 2012) are proposals for such languages.

3.4 Dependency management

In Paper B, it is stated that “the partitioning of the overall design problem in mechatronics into domain-specific problems leads to a situation where product properties influence each other, hence giving rise to dependencies” (Qamar et al., 2012). It is vital to manage dependencies for effective decision making. Considering the state of the art approaches, Paper B discusses whether dependencies could be captured in a general purpose systems modeling languages such as SysML (Object Management Group, 2012) or UML (Object Management Group, 2011). Both these languages are also supported in the Eclipse Modeling Framework (EMF) (Eclipse Foundation, 2009a), hence providing added advantages of using the services available through EMF to model dependencies. However, as per the recommendations in Paper B, at present, SysML does not provide sufficiently expressive language constructs to model dependencies.

The Design Structure Matrix (DSM) (Eppinger and Browning, 2012) is a well-known method proposed to capture relationships or dependencies during a design process. Some examples of applications of DSM can be found in software engineering (Sangal et al., 2005), and in product development (Danilovic and Browning,
Section 4.8 offers a detailed comparison between DSM and the dependency modeling approach proposed in this thesis.

Another popular approach towards dependency modeling is the Process Integration Design Optimization (PIDO) approach. ModelCenter (Phoenix Integration, 2012), modeFRONTIER (ESTECO, 2012) and Comet WorkBench (Comet Solutions, 2012) are well known tools in this area. For instance, through ModelCenter, a workflow process can be created by integrating disparate models. ModelCenter provides out-of-the-box connectors which enables data exchange with different tools. As a result, dependencies between properties existing in a range of models can be modeled. However, PIDO tools only provide a black box view for each model, meaning that the semantic relationships between properties are not shown. There is also an issue of models becoming too complex for understanding by a human since large numbers of dependencies typically govern evolution of product models. This issue is discussed further in Section 4.5.4.

PDM/PLM systems are also employed for modeling dependencies. They provide information about correspondence relationships between model elements, e.g. to maintain correspondence between a CAD model artifact and a system model artifact. However, these correspondence relationships are currently only modeled at an abstract level, and detailed dependencies between properties are not addressed by PLM systems.

Contract-based design is another school of thought relevant for dependency management. Sangiovanni-Vincentelli et al. (2012) define a contract \( C \) to explicitly handle pairs of properties, which represent both the assumption \( A \) of the environment and the promises \( G \) of the system under these assumptions, formally described as \( C = (A, G) \). Both \( A \) and \( G \) are satisfied by the set of all inputs and all outputs of a design. For example, for the water flow control system presented in (Sangiovanni-Vincentelli et al., 2012), inlet pressure \( P \) is defined as input, and water level \( \text{wl} \) and flow rate \( F \) as output. A contract can then consist of constraints defined on \( P, \text{wl}, \text{and} F \), which should be satisfied by the implementation (interaction of controller, sensors, valve etc.). Finally, an overall conformance to system specifications can be verified by deriving a composite contract - which is the composition of all the contracts of the components. This essentially means putting together all the equations in a system, so that they are all satisfied simultaneously. In (Derler et al., 2013), the authors apply a similar approach by describing contracts between control and embedded software design, e.g. related to scheduling, execution time, delay etc. Again, the contracts are derived based on the assumptions (e.g. about the environment) and promises that the controller and embedded software (together) should deliver. By complying with the contract at all times during the design process, the control engineer and embedded software developer can manage dependencies that influence their work. Similarly, contracts can be specified between other design domains, such as mechanical design and controller design, mechanical design and FEM analysis.

This thesis proposes an approach for dependency modeling which targets the explicit modeling and management of dependencies between model data through the
constructs of an expressive Dependency Modeling Language (DML). The resulting *Dependency Model* (also referred to as Dependency Network)\(^1\) presents information about the known dependencies between properties. In essence, the dependency modeling approach proposed in this thesis and design-contracts are related, although we believe that they attack dependency management problem from different perspectives. Section 5.2 describes the possible connection between these two approaches.

### 3.5 Consistency management

Due to the dependencies between design activities in different domains, an issue of consistency between the corresponding design models arises. Changes in one model may lead to necessary changes in properties in other models. However, such a change cannot be administered without knowledge of the dependencies. Most of the work focused on consistency between design models originates from the software engineering discipline. The topic of consistency between mechatronic design models has recently received some attention within the mechatronics research community. The types of consistency issues that are dealt with in the current literature can be split into two categories. The first category deals with consistency within the model itself (Herzig et al., 2011). Approaches such as Mens et al. (2005, 2006) and Simmonds and Bastarrica (2005) are examples of this category, where authors propose using a description logic to manage consistency of UML or SysML models. However, the presented work is not complete and needs further development.

The second category deals with consistency across disparate models. For instance, Adourian and Vangheluwe (2007) show how to relate two different modeling languages through their corresponding meta-models. An *association model* takes care of relationships between model elements of both languages. A bi-directional transformation is then used to ensure consistency, such as between Solid Edge (Siemens PLM Software, 2011) and UML models as illustrated by the authors. Gausemeier et al. (2009b, 2007) explain an approach for managing consistency between a domain-spanning principle solution, and domain-specific models through triple graph grammars (TGG) (Schurr, 1995). Automated model transformations from the principle solution to the domain-specific models support the evolution of models during transition from conceptual phase towards embodiment and detailed design phases. However, for both the Adourian and Gausemeier approaches, modifying or deleting individual components could still lead to inconsistencies among models.

Hehenberger et al. (2010) discuss an approach for instant consistency checking based on a solution called *model analyzer*. The analyzer detects any change to a design model and triggers a consistency checker, which evaluates the relevant consistency rule for the performed changes. The authors have argued that this

---

\(^1\)Dependency Model is also referred to as Dependency Graph, although the Graph might contain less detail than the Model.
3.5. CONSISTENCY MANAGEMENT

Success factor Difficulty in assessing consequences of choosing between alternative concepts
Transfer of models and information between domains
Lack of a common language to represent a concept
Level of consistency between design models
Level of complexity during integration phase
Ease in finding best design concepts

Figure 3.2: Reference model. The plus sign reflects a positive effect (e.g. an increase in level of complexity). The minus sign reflects a negative effect (e.g. a decrease in level of consistency).

approach provides support for maintaining consistency between models developed across different domains. However, it is difficult to define the consistency rules for all possible scenarios, and some consistency rules can only be defined in an abstract manner. Although this approach is useful in solving a subset of the consistency problems that arise during the mechatronic development process, the topic of consistency management needs further attention.

By proposing an explicit model capturing the known dependencies for a mechatronic product, we argue that it is possible to support consistency checking - for instance in cases when the value of a property changes. The reader is referred to Section 4.5.2 for further details.

Based on the reviewed literature, the current situation (in the design of mechatronic products) can be represented through a reference model (as explained in (Blessing and Chakrabarti, 2009, Chap. 4)). The three most important challenges that have led to the situation faced today are shown as blue boxes in the reference model (Figure 3.2). In order to identify the best design concepts for a mechatronic product, it is important that a designer can modify the product design dynamically, and be able to assess the consequences of different design decisions when assessing the alternatives. Therefore, we conclude that a core goal for the research reported in this thesis is the ease in finding the best design concepts, and a measurable success factor is the ability to detect errors, and thereby increasing the reliability of the product.
3.6 Conclusions on literature review

The comprehensive literature review in descriptive study 1 leads to the following conclusions:

- Different solutions have been proposed in literature to aid mechatronic design. However, many challenges related to the design of a mechatronic product are still not fully addressed. Three of these challenges were highlighted based on the investigation in Paper A.

- One of the common factors behind each of the challenges is a lack of dependency management.

- Dependencies within mechanical design need to be considered in order to accommodate local design changes; however, integration with mechanical design tools has historically been difficult. Therefore dependency management for mechanical design becomes even more challenging.

- Different approaches to ensure consistency between models have been proposed, but they are not complete. Dependency modeling can play a vital role in ensuring consistency.

Based on the above conclusions from the literature review, it was possible to make a decision about the potential areas in which research efforts should be focused during the prescriptive phase. The impact model in Figure 3.3 visualizes this by showing the intended support, and how it changes the situation depicted earlier through the reference model in Figure 3.2. The intended support is based on the provision of a dependency modeling language and a supporting tool, which enables modeling and management of dependencies during the design process. Furthermore, methods for enabling communications between models will also be sought. The predicted influences of the developed support on the success factors are included in Figure 3.3.

3.7 Chapter summary

This chapter provided a brief overview of the two descriptive phases, where the current situation was deduced through the literature review. The state of the art - relating to mechatronic design, integration between mechatronic domains, dependency management, and consistency management between design models - is presented. Based on this literature review, the reference model was developed to determine the factors that have led to the current situation in mechatronic product development. These investigations aided in deciding the areas in which support should be developed during the prescriptive study phase, and led to the creation of the initial impact model.
3.7. CHAPTER SUMMARY

Communication between models
A Dependency modeling Language and supporting tool to model dependencies
Level of consistency between design models
Level of complexity during integration phase
Support

Time to develop
Reliability
Ease in managing an engineering change order
Ease in finding best design concepts

Measurable success factor
Success factor

Figure 3.3: Impact model.
Chapter 4

Dependency Modeling and Model Management

This chapter explains the investigations performed on modeling and management of dependencies during the design of mechatronic products. In Chapter 1, we have discussed how dependencies play a key role during the design process. A model-based design process promises better complexity management, improved design quality, better knowledge reuse and improved communication (Friedenthal et al., 2008). However, the challenges such as model management, data exchange, model interoperability, and consistency management need to be addressed in order to gain value from a model-based effort. We have talked about how the inconsistency between design models could be a potential calamity. The aim is to equip ourselves with the best possible methods and tools that can support integration between disparate models, and help in detecting and avoiding as many inconsistencies as possible. This chapter will highlight one such method to manage consistency, i.e. through dependency modeling. In addition, the chapter also highlights the use of model transformations to support integration and communication between disparate models, exemplified by transformations between SysML and CAD models. The work reported in this chapter is mostly based on Papers B, C and D, with initial ideas reported in Papers E and F. The reported work has a direct relation to RQs 1, 2, 3 and 4, along with providing partial answers to RQ 5 and 6.

4.1 Decision-based engineering design

Before we look at the dependency management problem, it is important to build an understanding of how inconsistencies appear, and what type of decisions can lead to them. In order to do that, we will first look at the role of decisions in a design process, and then build a classification of consistency problems.

Design can be considered as a decision-making process. Based on this notion, and building upon the theories of decision making and utility, Hazelrigg (1998)
proposed a framework for decision-based engineering design. In the following, we will utilize the ideas presented in (Hazelrigg, 1998, 1996) to build an understanding of two terms: ‘rationality’ and ‘value’. This will enable us to describe what kinds of decisions are rational, and how to focus on value during a design process. The reader is alerted that when the word ‘option’ is used (in the text that follows), it refers to a possible design solution or a design concept for a product.

According to Hazelrigg, engineering design only entails performing two main steps:

• Exploring the design space for possible options.
• Selecting the best option.

As simple as it seems, both of these steps are rather difficult to perform. For instance, for any product, the range of possible design options is almost limitless. Although the product specifications limit the design space, obviously it is still not feasible to find all possible design options as that number is infinite. The decisions we make in finding the possible design options also carry a degree of uncertainty since it is not possible to fully know how a particular design option performs until it is actually built. As engineers, we utilize models to make the best possible prediction about how a particular option will perform. However, models are only abstractions of the real world, and cannot predict the future with certainty. Hazelrigg suggests that to rank order design options, it is important to have a valid measure of value for each option. However, due to uncertainty, it is non-trivial to identify a valid value measure. For a particular product, even if one finds a large number of possible design options, builds perfect models to predict their behavior, and constructs a valid value measure to rank order for every option, it is still not computationally feasible to evaluate all these options to find the best one. Which is why human input will always be required at many points during a design process, making decisions about which design variables to consider, or determining an appropriate value measure, and so forth.

Hazelrigg emphasizes that every decision we make as designers should be considered in the context of the value measure. According to him, decisions involve options, expectation, and value. In this sense, a preferred decision is the option for which the expectation has the highest value. All values are expressions of human needs, wishes and desires. The concept of value is the same as the concept of objective function in numerical optimization - both are used to rank order alternatives. Rank ordering all alternatives imposes stringent mathematical conditions on the preferences of the designer (Hazelrigg, 1998). For instance, consider a situation where a designer is confronted with three design options: A, B, and C, each with a corresponding known and deterministic outcome. The decision maker is asked to rank order the preferences with the result that A is preferred over B, and B is preferred over C. This can be represented as $A > B > C$. In such a preference ordering, there is a real scalar function $\mu$ called the utility, such that $\mu_A > \mu_B > \mu_C$. The utility could be regarded as a profit in a trade scenario, which means selling A
for B for a small profit, B for C for a small profit, and so on. Now, instead, if the designer chooses a preference ordering of $A > B > C > A$, then this would require a utility such that $\mu_A > \mu_B > \mu_C > \mu_A$. As $\mu$ is real, such preference ordering is clearly incorrect. In our trade scenario, this means that at the end of the trade, we are left with more than we started with. It is mathematically impossible to construct an objective function out of such preference ordering, which Hazelrigg labels intransitive, and the person who has such a preference as irrational. On the other hand, if the ordering $A > C$ applies for every preference ordering of the form $A > B > C$, then the preference is said to be transitive and the decision maker is called rational. Note that the ordering $A > C$ has to be applicable for all preference orderings in this case. If for a particular preference, the ordering is not applicable, then we will still call it irrational. The important point is that only rational preferences lead to mathematically consistent models, and by making rational decisions, designers seek to maximize the value, reflecting satisfaction of human desires and needs. We re-visit the consistency topic in detail in Section 4.3.

Based on Hazelrigg’s proposed design framework, designers first search the design space to find possible design options. By modeling, the designer expresses their beliefs and aims to predict each alternative’s outcome, which according to Hazelrigg is called an expectation. The designer then specifies their preferences to build up selection criteria, forming the basis of expected utility $E[\mu]$ (Figure 4.1). The aim of the decision making process is to maximize the utility value, and as a result, to find the most preferred alternative. The employed model-based design effort seeks to enable better decisions with the available resources (effectiveness), and also to try to make these decisions with fewer resources (efficiency). Note that a decision is good - if and only if - it leads to an increased value, in order to fulfill the objective of maximizing the value. If there is no focus on value, there is no way to find the effects of our decisions. As per (Hazelrigg, 1998), it is possible that engineers can build a sound mathematical formulation when given a problem to solve, however when given a decision to make, some tend to rely more on guesswork than on their knowledge. The role of modeling is to provide enough guidance to make a well-informed decision. Unfortunately, some look at modeling solely as a performance measure and forget to take into account the prevailing uncertainty.

Effective decision making requires consistent beliefs and rational preferences. But consistency could mean many things. For instance, irrational preferences could lead towards inconsistency with laws of mathematics. We explain the consistency classification later in Section 4.3.1. For now, we can deduce that a model is only useful if it is constructed with consistent beliefs. In addition, to have an overall value, the benefits of modeling have to outweigh the costs.

### 4.2 Properties and the decision making process

In order to dig deep into how dependency management is related to consistency, it is important to form a fundamental basis for properties and decision making. Rational
Design Theory (RDT) (Thompson, 2011) builds upon Hazelrigg’s decision-based engineering design foundation. This theory provides a conceptual model where design options are specified and predicted in terms of their properties, and decision-making criteria are defined in terms of these properties. The following paragraph is based on the development work performed in (Thompson, 2011), and we will utilize RDT to build a foundation for properties, property dimensions, design artifacts and the decision-making criteria.

Anything produced through human intelligence and effort is considered an artifact (Baldwin and Clark, 2000). Thompson described a property as any descriptor of an artifact, where any is used to refer to the infinite number of properties that can describe a given artifact. If $S'$ is a set containing all possible artifacts, e.g. $A_1$, $A_2$ and $A_3$, then a property is mathematically defined as a function over the artifact set $p : A \rightarrow Y$, where $A \subseteq S'$ ($S'$ is the domain of the property), and $Y$ is the topological space that constitutes its range (Figure 4.2). For instance, property mass is defined for mechanical artifacts, and its range could be a real positive number. The collection of the topological spaces of all properties is called the property space, $P$. Describing artifacts may require a large number of properties, leading to a multi-dimensional property space, where the number of properties is potentially infinite. However, for a particular context, not all the properties are relevant, and limiting to a finite set of relevant properties is therefore essential. Through this property set, the designer makes a decision about what is to be included in the model, and what type of questions the model will help to answer. This process is
4.2. PROPERTIES AND THE DECISION MAKING PROCESS

Figure 4.2: Artifacts and property spaces (Thompson, 2011).

called an abstraction. Thompson proposes that mathematically, this abstraction corresponds to a property space projection $P'$, which is the Cartesian product of the ranges of all the properties considered relevant. As a result, each artifact can be represented by a point in the property projection. However, the complete description of any artifact would require an infinite number of properties. Therefore, a point in the property projection is an abstraction of infinite number of possible artifacts. Specifications put further constraints on a set of properties, limiting the value that a particular property can take or is supposed to represent.

The property projection forms the basis of defining design concepts or design options. Thompson declares a concept $C$ as a partial specification of an artifact (mathematically, $C \subset P'$). She included the qualifier partial to suggest that regardless of how much detail is included in a concept definition, it is not possible for every property to be specified. Furthermore, a concept is a specification for a hypothetical artifact. Hypothetical because a designer considers a number of concepts before making a decision about a particular artifact. Hence, it is possible that a concept is not selected, and therefore never realized. Another reason Thompson declares the artifact hypothetical is because the concept may not be realizable, meaning that there may be no artifacts that satisfy all specifications within the property space. Limitations of technology may also render the artifact impossible to manufacture.

The process of developing a concept is hence an iterative one, where concepts are refined as further constraints are put on properties. In the beginning of the design process, product specifications are used to constrain a set of properties. For
instance, for a robot, the analysis of environment, or a specific requirement may dictate a decision that the robot should operate in a planar workspace (Herzig et al., 2011). This decision could lead to a concept $C_1$, which contains all possible robots that operate in a planar workspace. Further refinements of an initial concept may introduce more constraints. For instance, a concept $C_2$ can be selected as a two degrees-of-freedom revolute robot, and $C_3$ can be selected as a XY-table providing freedom of motion in two-degrees. Both these concepts fall under design options for operating in a planar workspace. The designer is now faced with a decision to determine which one of the two concepts to consider. As per Hazelrigg, the alternative that leads to the most preferred outcome is the concept to consider. To determine that, it is necessary to predict the outcomes of selecting $C_2$ and $C_3$ respectively. Thompson introduces a term here called a concept prediction: “a concept prediction $X$ is a mathematical characterization of the designers beliefs about the properties of the artifact that will be realized when the concept $C$ is used as specification” (Thompson, 2011). The concept could be either manufactured to find the outcome, or predicted through analysis models. As per Hazelrigg’s model (Figure 4.1), the concept prediction refers to the outcomes of a design alternative. Designers try to predict the behavior of a concept in order to verify the mapping to the possible artifact. However, as Hazelrigg suggested, this verification is subjected to uncertainty, and cannot be guaranteed. Therefore, uncertainty in models and in predictions is a challenge. The final step is to rank order the concepts as per their predicted outcomes and the designer preferences, in order to establish an expected utility $E[\mu]$. This step is referred to as concept decision criteria, which mathematically is an evaluation function $E$ that maps the probability measures $(X)$ onto the real axis (Figure 4.3). The real-value could be expected utility, or cost for a particular concept. Optimization techniques can be employed here to provide a more thorough evaluation depending on formulation of the evaluation function.

4.3 Fundamentals of consistency management

Now that we have established what types of decisions are rational, and how to focus on value while making decisions, we can now turn our attention to the causes of inconsistencies. Consistency refers to a state in which no contradictions are present. Contradictions could arise from many sources. During the design process, the product is still an unknown entity. Therefore, different methods are utilized in order to gain better understanding of the product, e.g., through models, through prototyping, and through testing. The process of using these methods is affected by the decisions designers make. In a model-based design effort, the model is treated as a primary artifact, which reproduces the properties of the object being modeled (Buur and Andreasen, 1989). The designer decides what is to be included in the model based on the type of questions the model should answer (abstraction level). The influence of a designer’s decisions can be explained through Tjalve’s (Tjalve, 2003) investigation on product development methods, where he states that
if two product alternatives carry the same basic structure, the variation of structural elements in the quantified structure influences the design greatly and would lead towards very different looking products. Through design methods, processes, and tools, we strive to provide designers with the best possible mechanisms to answer questions about the product under design. We also aim to ensure that the performed design and analysis is less prone to errors.

The path that designers take when searching for design solutions for a product is based on how quickly they are able to increase their knowledge about the product, their beliefs and preferences. In this regard, any design decision could lead to contradictions. The question is how to identify whether there is a contradiction between the proposed design concepts and the final product? Answering such questions requires the designer to build a conceptual foundation, not only to define consistency, but also to characterize the different types of inconsistency that can occur during the course of the design process (Herzig et al., 2011). After building a classification for inconsistencies, the next goal is to investigate whether they can be rectified or not.
Figure 4.4: Illustrating external and internal consistency.

4.3.1 Classification of consistency

In (Herzig et al., 2011), the authors laid down a conceptual framework for managing consistency, providing two main classifications: external consistency and internal consistency.

External consistency

The information about a design artifact could be captured formally or informally. For example, sketching a design concept is an informal form of modeling, which is based on visualization of the concept in the mind of the modeler. Whether this design concept is consistent with the laws of nature and the actual object in the real world is classified as a question of external consistency (Herzig et al., 2011). External consistency applies to the reality in which the product will ultimately exist. For mechatronic systems, this refers to the real-world, where laws of nature such as of physics are in effect (see Figure 4.4). However, our understanding about the laws of nature is not perfect, and as such can be treated as mere belief rather than accepted as the actual laws themselves. Therefore, the designer’s beliefs are subject to an amount of uncertainty. Furthermore, the knowledge about the product increases during the design process, and as a result the final product is not known until it is actually built. Hence design concepts will never represent all the details of the final product (to be) and are subject to inconsistencies which ultimately cannot all be checked for, as per (Herzig et al., 2011).

Internal consistency

Internal consistency is related to the situation where systems are developed under certain well understood axioms such as logical and mathematical rules (Herzig
et al., 2011). Modeling languages are constructed based on these rules. Hence, the grammar of the modeling language must be consistent with the rules of logic and mathematics. Furthermore, the models developed through these modeling languages should reflect the proper use of the language constructs in order to be internally consistent. *Logical inconsistencies* may arise by not adhering to the rules of the modeling language. This can occur if a modeling language does not strictly enforce the proper application of the rules, as with semi-formal modeling languages such as UML and SysML (Broy et al., 2010). Tools that support such modeling languages cannot enforce strict adherence to the logical rules of the language, thus not all logical inconsistencies can be detected.

In addition to logical inconsistencies, failure to adhere to the laws of mathematics (while modeling) will lead to *mathematical inconsistencies*, such as failure to adhere to Kolmogorov’s axiom of probability. For example, during a fair coin toss, if it is assumed that the probability of heads is 0.5, and probability of tails is 0.6, then the assumption is internally inconsistent with the laws of mathematics. On the other hand, if the probability of heads is 0.3, and tails 0.7, leading to a total probability of 1, then the assumption does not violate Kolmogorov’s axioms and is internally consistent. However, this concept will be externally inconsistent with the laws of nature for a fair coin toss, where the observations demand equal probability of heads and tails. A modeling language, even if adhering to the laws of mathematics, may allow development of models to be internally consistent but externally inconsistent at the same time.

Our understanding of the laws of nature is constantly evolving, hence there could be inconsistencies due to our beliefs at any given time. Further inconsistencies may occur while making decisions. As discussed earlier, irrational preferences could lead to *intransitive preference* ordering, and it is mathematically impossible to construct an objective function from such preferences. Such decisions are clearly irrational, i.e. they are inconsistent with rationality, and are labeled as *inconsistencies due to preferences*, to differentiate them from *inconsistencies due to beliefs*. Only rational decisions lead to consistent design concepts. Figure 4.4 illustrates the type of consistencies in relation to a real-world artifact.

### 4.3.2 Consistency during model-based mechatronic design

In this section, we aim to briefly illustrate the classification of external and internal consistency through a fairly simple robot design example. The best practices of Model Based Systems Engineering (MBSE) were followed for modeling the robot. The robot design begins with an initial set of requirements. Figure 4.5 shows some of the requirements formally represented through a SysML requirements diagram. Some of the requirements in Figure 4.5 include the requirements on workspace, Degrees-Of-Freedom (DOF), and the position error. From the requirements, the design activity leads towards a functional view as shown in Figure 4.6. As per Tjalve’s design principles (Tjalve, 2003), to fulfill the functions and the set of requirements, a basic structure for the robot was chosen, namely two mechanical
Figure 4.5: Top-level requirements for the robot.

arms, two joints - each actuated by a motor, and two sensors - each measuring the amount of rotation of one joint. The basic structure is shown as a SysML block definition diagram in Figure 4.7. Based on this basic structure, two quantified structures, which satisfy the requirements, are selected. The first quantified structure includes equal length of both arms, and the other includes a longer first arm and a shorter second arm. In addition to SysML for system modeling, Matlab/Simulink (Mathworks, 2013) and Solid Edge were used to refine the design concepts and aid in the decision making process.

The quantified structures are evaluated against a set of basic properties (designer preferences, such as reliability, manufacturing cost, and controllability) and the aim is to predict the basic properties in order to assess the two alternatives. For example, the reliability of the robot is the product of the reliability of each degree of freedom, which is the product of the reliability of individual components i.e. the arm, the joint, the motor, and the position encoder. The motor and the actuator are typically off-the-shelf components, with the reliability figures provided by the manufacturer. For the arm, which for this example is a manufactured component, a stress and strain test can be performed to select a confidence level above 99%. However, when a motor fails, it does not only lead to the malfunction in one degree of freedom, but makes the whole robot unusable for further operation. Furthermore, when specifying reliability, consistency with Kolmogorov axioms, and hence the laws of mathematics must be ensured. For example, specifying the probability of success of the robot to be 99.999%, while declaring the probability of failure within the
4.3. FUNDAMENTALS OF CONSISTENCY MANAGEMENT

Figure 4.6: Robot functional view represented though SysML use case diagram.

same time frame to be 10% is clearly an inconsistent specification. By predicting the basic properties considered by the designer, it is possible to make a decision about selecting one of the two concepts. However, it is clear that this decision is only well-informed if the designer’s beliefs and preferences are rational, and the developed models do not break the rules of mathematics or logics.

SysML also allows modeling of behavior aspects of the robot, leading to different views inside a SysML model. Internal consistency requires all these views to be consistent with each other, i.e. requirements consistent with function specifications, functions with structure, structure with behavior, and behavior with requirements. Furthermore, each of these views should be constructed by strictly adhering to the SysML grammar, so as to be logically consistent.

The product properties are spread across disparate models developed through different modeling languages, and it is common that these properties influence each other, referred to as dependencies. For instance, controllability is dependent upon the mechanical properties of the robot, while the workspace of the robot is dependent upon the geometry of the physical structure. A mechanical design that satisfies workspace requirements may at the same time have limited controllability, and a controller design may satisfy the controller requirements but only with a suggested change in mechanical design. Therefore, apart from a model being internally consistent, it is also important that the information is consistent across different models (Herzig et al., 2011). The dependencies are spread across the domain-specific models used in design and decision making, and mismanagement of these dependencies can lead towards internal inconsistencies.
We can conclude this section by reaffirming that a model is only (fully) valuable if it is consistent. Inconsistent models could lead to making wrong decisions leading to potential failures and calamities. Moreover, a model is only valuable if its benefits outweigh its cost. This is essential since *modeling too expensive* is clearly not the aim during a product development effort. To study the consistency management problem, we will first investigate how to manage dependencies? We will also try to answer the question: is dependency modeling valuable? It is important to mention here that only internal consistency is considered for this work; we have deduced that it is not possible to check for external consistency. In the following, we will explain the fundamental concepts behind dependency management and relate them to managing internal consistency.

### 4.4 Fundamentals of dependency modeling

Ensuring consistency across models is a huge challenge. Model-creating humans are the prime suspect when it comes to inhibitors to consistency management. By introducing change to a model, or by not conforming to the rules of the language,
simply by holding conflicting beliefs, the human factor, whilst essential, is unpredictable. By capturing the product information formally across a set of models, the modeler expresses their beliefs about a number of product properties, which are then spread across a set of models. As previously discussed, we use the term *dependency* to indicate that a value of a property depends upon the value of another property. As the product properties are spread across a set of models, there are obviously dependencies between them. These dependencies are usually only implicitly known, meaning that engineers possess knowledge about dependencies, which are never captured formally or explicitly in a model. It is difficult to deduce the impact of a change introduced in one model on the other models in the set, resulting in inconsistencies (Qamar et al., 2012). The focus of dependency modeling is to study the role of dependencies at a fundamental level, and investigate to what extent it is possible to capture the dependencies. This section is mainly based on the work reported in Paper B, and revolves around the following hypotheses:

**Hypothesis:** Making dependencies explicit facilitates consistency management and adds value.

In order to test the hypothesis, the following questions are asked:

- How detailed should the dependency representations be?
- How can dependency models be managed, and how will humans utilize these models?
- What type of support is available in current modeling languages to model dependencies explicitly?
- How can the value gained be quantitatively compared against the effort required in creating and managing dependency models?

In the current literature, a few solutions are proposed to manage dependencies, e.g. through DSM (Eppinger and Browning, 2012); through correspondence graphs supported by model transformations such as in (Cao et al., 2011; Shah et al., 2010); or through Domain-Specific Languages (DSL) such as SysML profile for Modelica (Paredis et al., 2010). However, most of the solutions either focus only on implicit management of dependencies, or fail to consider that dependencies could be captured at different levels-of-detail (Qamar et al., 2012). In terms of consistency checking, most of the proposed solutions are ad-hoc approaches, for example consistency checking of UML models (Mens et al., 2005) and maintaining consistency across domain-specific models such as (Gausemeier et al., 2009b; Adourian and Vangheluwe, 2007).

### 4.4.1 Properties

In design, we use properties to describe constraints (*specification*), or - given a particular specification - for communicating the designer’s belief regarding the value
of the property \textit{(prediction)}. Properties can have a logical or numerical nature. For example a car’s property \textit{having an engine} to be true to false, or property \textit{horsepower} with a value of 300 hp.

We distinguish between synthesis and analysis properties. Synthesis Properties (SP), or specification properties, are used to define system alternatives. The specification of a system alternative consists of a set of constraints imposed on SPs, which are typically specified in terms of property ranges - for instance tolerance constraints on the specified geometry. Multiple SPs together represent a specification of a system alternative considered in a design decision. Through analysis and optimization the designer will ultimately choose one of the specified alternatives as the most preferred. A system alternative could also be specified parametrically through SPs, i.e. through some SPs which are freely chosen and other that are defined as parametric functions of the freely chosen ones. For the robot with two mechanical arms (Figure 4.7), SPs could be lengths and widths of the links.

Analysis properties (APs) capture the beliefs of the designer and thus constitute predictions rather than specifications of system alternatives. For instance, given a specified geometry (using SPs), the designer may predict the cost or the mass of a component - cost and mass are APs in this example. Since APs are predictions about the future, which is inherently uncertain, they should be expressed in terms of \textit{probabilities} or probability density functions (as opposed to property ranges for SPs). APs are often computed by using predictive models which mathematically capture the beliefs of the designer. Although it is not absolutely necessary for the mathematical relationship between APs and SPs to be modeled explicitly, the APs must be kept \textit{consistent} with the SPs that define the corresponding system alternative. Keeping the robot with two mechanical arms (Figure 4.7) in mind, example APs are inertia of arm, cost of the robot and rise time of the closed loop system. Each of these APs is predicted by using analysis models.

For a property to have an unambiguous meaning, it is not enough to specify whether the property is an AP or an SP, the \textit{semantic context} around the property needs to be defined. This context is established through relationships with \textit{concepts}, giving rise to a network of properties. For instance, a system alternative for a car can be defined through the following constraints on SPs: 4 wheels, an engine, engine with 4 cylinders, cylinder bore diameter 0.100 +/- 0.0001m. The SP for the \textit{diameter of the cylinder} is the \textit{value} associated with the concept “Cylinder”, i.e. bore diameter is a value property of the concept cylinder, which in turn is a part of the engine, which is a part of the car (Figure 4.8). Through semantic relationships such as “part of” or “value of”, the SPs and APs are ultimately networked to each other. Similarly, the APs specifying a prediction of a system alternative also need to be related to the network of SPs, which provide the semantic context for APs.

It is possible that properties are parts of the same semantic context without affecting each other’s value. For instance, in Figure 4.8, an \textit{Engine} is a part of a \textit{Car}. At the same time, a pair of \textit{Seats} can also be a part of a \textit{Car}. Now properties of the \textit{Engine} and the \textit{Seats} do not affect each other’s value, however, they are semantically related.
4.4.2 Dependencies

Apart from specifying the type of properties and their semantic relationships, we use the term dependency to indicate that properties are dependent upon each other, e.g. a mathematical relationship expressing how an AP depends upon an SP. We make a distinction between two types of dependencies. The first type of dependency is about the choice made by the designer, called Synthesis Dependency (SD). The choice is in terms of selecting the value of an SP. For instance, using controller design heuristics to select controller gains - controller gain is an SP in this case and the heuristics used represents the SD. As pointed out earlier, it is possible that an SP is a parametric function of other SPs which are freely chosen, which means that an SD can also be represented by a parametric computation. Since SDs may reflect heuristics, it is possible that a human overrides the heuristics while selecting the value of an SP. The choice could be based on new experiments, observation, or recent experience, rendering the heuristics inapplicable. In this case, even if the value of SP is not consistent with the heuristics, the choice (made by the designer) is still considered consistent with the current SD, and the override is modeled as a cross on the representation of the SD, as shown in Figure 4.9. The cross depicts a decision point controlled by a human.

Figure 4.8: Building a semantic context around a property.
The second type of dependency is called Analysis Dependency (AD), which represents predictive models used to predict an AP. The modeler expresses their beliefs in the model by building mathematical relationships between a set of SPs and APs. For instance, when predicting the rise time of the closed loop system by selecting controller gains, the analysis used to predict the rise time represents the AD in this case. Another example is the analysis provided by a CAD tool to predict mechanical properties. It is not necessary for an AD to have a mathematical representation, the mathematical formulation might not be known and it may be enough to formulate that an AD exists. A discussion in Section 4.4.5 on level-of-detail in dependency modeling elaborates this further.

One might ask “is it possible that a property is an SP at one design stage and an AP in another?” The answer is “No”. What is possible is the existence of more than one property associated with a single physical quantity. For instance, at an early stage of design, one may choose that the desired accuracy of the robot is 0.5 mm, an SP. In a subsequent stage of the design process, one is likely to revisit the accuracy issue at a more detailed level, and now determine the predicted accuracy based on the choices made for the selected geometry, controller parameters etc. This predicted accuracy is a different property - now an AP - associated with the same quantity “accuracy”. Typically, the designer will select the robot details so that the predicted accuracy is within the desired accuracy.

4.4.3 Dependencies across models (in different domains)

In mechatronic design, design activities are spread across different domains such as mechanics, electronics and software. The modeling activity performed in each domain leads towards the creation of domain-specific models. Within a domain-specific model, there are dependencies (SDs and ADs) between SPs and APs. For instance, the analysis supported by a mechanical CAD tool constitutes an AD to predict the mechanical properties of an object. At the same time, there are dependencies in between domain-specific models. For instance, an AP in one domain-specific model can become an AP in another model, or in other words, the two
APs are bounded by an equality dependency. This is illustrated in Figure 4.10 where two domain-specific models, Model A and Model B, are shown. Based on the chosen SPs, Model A provides a prediction of an AP, which is bound to an AP in Model B. A similar situation can happen in the case of SPs. Such cross-domain equality dependency bindings could be numerous, leading to a multitude of properties being shared between two or more models. In this case, instead of a single property binding, it is an Analysis Property File (APF) or a Synthesis Property File (SPF) containing a list of properties being shared between models.

Apart from property binding across two models, there could be other scenarios of dependencies between domain-specific models. For instance, a situation of property refinement where multiple selections refine the same SP as shown in Figure 4.11(left). For the robot example, it could be the geometry of an arm, which was initially chosen through CAD modeling, and later a different geometry is selected after the FEM analysis has been performed. Therefore, there are multiple selections for the length of the arm and the width of the arm, with one refining the other as per Figure 4.11(left). Figure 4.11(right) shows an alternative case for an AP, i.e. each model predicts an AP. In this case, a new AP could be a mathematical function of individual APs. Both these scenarios represent dependencies across domain-specific models.
Figure 4.11: (left) A set of models influencing the same property, in this case a SP, and (right) analysis properties predicted from Model A and B being used in Model C to predict a new AP.

In the current practice in MBSE, the dependencies are only implicitly known. Based on the terminology presented for the SD and the AD, it is possible to have an explicit formal representation of dependencies. In this case, for an AD, a reference to the analysis model or an equation is maintained, and for an SD, a reference to the heuristics, guidelines or a mathematical formulation is maintained.

4.4.4 Illustration

We will now turn towards illustrating the dependency modeling terminology through a design example. The problem at hand is to pick and place an object in a three-dimensional environment with known obstacle locations. For simplicity, we have considered motion only in two dimensions, ignoring the movement in the third dimension. For this design problem, two alternative concepts were proposed. One based on an XY-table and another based on a two degree-of-freedom revolute robot (Figure 4.12). As per Figure 4.3, these two concepts fulfill the function of picking and placing an object inside a two-dimensional environment. In this illustration, our focus is not on illustrating how the concept of an XY-table is valued against the concept of a revolute robot. Our focus is rather on illustrating how dependency
modeling is performed for a design concept.

As per Tjalve (Tjalve, 2003), the robot design process begins by synthesizing the basic structures, one of which is chosen to synthesize quantified structures. In Figure 4.13, one possible quantified structure of the robot is shown, along with the workspace, the pick and place point, and the obstacle location. There is a requirement that the robot position should be controlled with certain accuracy, here called Controlled Position Accuracy (CPA). Some design variables considered part of the specification include: link length \( (L_A, L_B) \); link width \( (W_A, W_B) \); material density \( (\rho) \); range of joint motion \( (\theta_A, \theta_B) \); point of origin \( (O) \); torque of motor \( (M_A, M_B) \); resolution of sensor \( (S_A, S_B) \); and maximum distance \( (PE) \) between pick and place point. It is assumed that the robot can grip objects of any size; hence no gripping movement is considered in this example.

In the following, we will consider two design domains for the robot: mechanical design and controller design. Mechanical design is performed to make sure that the robot meets the Work Space (WS) requirement, as well as predicting the mechanical properties of the robot. Knowing the mechanical properties, a kinematic analysis is performed in the controller design domain to predict the End Position (EP) of the
robot based on joint angles. Based on the required CPA, Controller Gains (CG) are selected to satisfy the accuracy requirement for the desired robot movement.

The mechanical design of the robot is performed in the CAD tool Solid Edge (Siemens PLM Software, 2011), and Matlab/Simulink (Mathworks, 2013) is used for dynamic analysis and controller design. As soon as the CAD model and the Simulink model are created, the dependencies between the two can be modeled by following the terminology presented in Section 4.4.2. The dependency model will - in this case - show the dependencies within and between the CAD and Simulink models. Figure 4.14 shows the resulting dependency model, with SDs and ADs in orange, SPs in blue and APs is green. The rectangles showing mechanical design and controller design signify the CAD model and the Simulink model respectively. As an example, consider the $SD_1$ where $WS$ (work space) and the distance $PE$ lead to a selection of link lengths ($L_A$ and $L_B$). This indicates that both the link lengths’ are selected by the designer based on the specification for work space and the distance between pick and place point. It has to be noted that $SD_1$ exists
4.4. FUNDAMENTALS OF DEPENDENCY MODELING

within the CAD model. For the case of property binding, consider $AD_3$, which represents the kinematic analysis modeled inside Matlab/Simulink. To perform the kinematic analysis, $AD_3$ requires geometrical information for the robot from the CAD model. Therefore the properties link length ($L_A$ and $L_B$) and origin point ($O$) of the robot have an equality binding (same as in Figure 4.10) to $AD_3$, which results in prediction of the end position ($EP$).

The dependency model shown in Figure 4.14 contains a network of properties and dependencies. Therefore we use the terms dependency network or dependency
Inertia

Figure 4.15: Illustration of causality. (a) Analysis 1 predicts Inertia by selecting geometrical properties a, b, and c. (b) Analysis 2 predicts a, b, and c by selecting a known value for Inertia.

graph while referring to the dependency model. It is important to note that the dependency graph is a directed graph, i.e. the causality cannot be reversed. Let us consider Figure 4.15 to explain this. In Figure 4.15(a), an analysis model predicts a property Inertia once the material is chosen and the geometrical properties a, b, and c are selected by the designer. Analysis 1 is deterministic, since specific values of a, b, and c always leads to a specific value for Inertia. In Figure 4.15(b), an analysis model (Analysis 2) predicts the same properties a, b, and c once the property Inertia is selected. However, Analysis 2 is non-deterministic since several combinations of a, b, and c could be found for a given value of Inertia, and the whole design space has to be searched in order to find these solutions. In engineering design, Analysis 2 is considered as Synthesis, which as per the Theory of Logical Reasoning [Peirce 1958], is Abductive. Whilst Analysis (such as Analysis 1) on the other hand is Deductive. For given starting values for a, b, and c, a round trip through Analysis 1 and Analysis 2 may lead to final values which are different than the starting ones. Therefore, the dependency graph is strictly causal. One could think that an acausal graph could provide advantages, for instance using a single analysis to either predict Inertia or a, b, and c in this case. However, developing such acausal analyses is more challenging.

It is possible that algebraic loops exist inside a dependency network, where a property could both be chosen and predicted. Figure 4.16 illustrates such a situation, where the accuracy property CPA is shown twice. One of the CPAs (in blue) is an SP which is constrained by the specification. The other CPA (in green) is an AP, which is predicted by selecting sensor resolutions S_A and S_B (based on the selected CPA), and propagating the mechanical properties through a kinematic model (AD4) to predict the end position EP. It is now possible to find the predicted CPA by finding the error between the predicted position and the reference position (AD5). The loop is repeated until the predicted CPA is within the bounds of the specified CPA. The situation illustrated in Figure 4.16 is common to the process

---

1A dependency graph may also be a partial representation of the dependency model.
of optimization where the objective is to optimize a parameter (in this case \( CPA \)) by selecting some design variables. Solving algebraic loops requires simultaneous solving of a system of equations through a technique called “tearing”. Tearing algorithms break the loop and perform iterations on reduced number of unknowns to find a solution (Elmqvist and Otter, 1994). In the case of a dependency network, SDs contain a decision point controlled by a designer, therefore it is most desirable to tear the loop at an SP - for instance, in Figure 4.16, we can tear the loop after \( SD_6 \). The crosses after \( SD_6 \) show tearing or override by the designer.

The reader should note that the illustration in Figure 4.14 is provided solely for the purpose of understanding. In reality, such a network will be created through a modeling language such as SysML, or a dedicated dependency modeling language. We discuss the Dependency Modeling Language (DML) in Section 4.9.

### 4.4.5 Modeling dependencies at different levels-of-detail

Dependencies can be formally represented at different levels of detail in a dependency model. Which level of detail should be included in the model depends upon the problem in hand and the context. In particular, the dependencies play a bigger role in large design problems, where big organizations are involved, hence manual management of dependencies becomes too labor-intensive and error-prone. In such cases, dependency modeling at a higher level of detail is likely to be most valuable.

Based on Paper B, we propose the following levels-of-detail as guidance.

**Level 0:** At this level, no explicit dependency modeling is performed, and dependencies are only known implicitly.

**Level 1:** At this level, the existence of a dependency is modeled, but without specifying what the dependency is; for example, formally stating that properties of an analysis model depend upon a CAD model. By expressing level 1 dependencies (Figure 4.17), it is possible to maintain traceability, such that a designer is reminded of which other objects to investigate/update in a change management scenario. It
also aids in analysis of a dependency network, e.g. to determine the optimum workflow for activities. For instance, based on the structure of the dependency network, one could break it down into parallel workflows for concurrent engineering. However, in many cases, it is not enough to know only that a dependency exists, since managing that dependency requires specific information about the actual dependency (level 2). Typically, PLM/PDM systems capture logical relationships between modeled properties at level 1, where they provide information about which model elements are related to each other.

Level 2: Dependencies become white boxes showing both the existence of the dependency and what the dependency is. An example is modeling correspondence relationships between properties of a system model and an analysis model as shown in Figure 4.18. The dependencies could potentially be executed and maintained automatically to reduce the effort, and to avoid inconsistencies. For instance, through a model transformation, a ModelCenter™model (Figure 4.19) could be built based

Figure 4.17: Level-1 dependency modeling, adapted from (Shah et al., 2010).
on the dependency model in Figure 4.18. The analyses (embedded in Matlab and Excel) could then be executed to find the property values, and automatically update these values to the system model, hence resulting in fewer inconsistencies. The challenge is that models exist in different tools, i.e. there is a tool-integration problem.

Figure 4.18: Modeling dependencies between system properties of a motor and its analysis in Matlab and Excel (Paredis, 2012).

**Level 3:** A model is only valid under certain conditions. A cantilever beam bends under a load, and the bending can be explained through a linear model assuming a small load. A larger load causes higher bending which is better explained through a non-linear model, and the linear model is no longer valid. Therefore, apart from modeling dependencies, it is also required to mention when the
Figure 4.19: A ModelCenter model created through a transformation based on the dependency model in Figure 4.18 (Paredis, 2012).

dependency is applicable, which corresponds to dependency modeling at level 3. The applicability conditions could be described through constraints on properties (which can be thought of as dependencies between properties and validity conditions), meta-data, or a validation data set. A design change can result in violation of an applicability condition, and such a violation can be discovered once the applicability conditions are formally specified. Therefore, modeling dependencies at this level of detail provides an added advantage of alerting the human (user) if the applicability conditions are violated.

**Level 4:** An important question that comes to mind while considering dependency models is: what happens when the structure of the model changes? For example, adding a third link to the robot in Figure 4.13. For the robot case, it is possible to deduce the change in dependency network by knowing what objects
were created for the previous two links. The knowledge of dependencies for the previous two links will in this case guide us in finding the required changes when the third link is added to the robot, a concept called a dependency pattern. Dependency modeling at level 4 includes modeling the dependency patterns, which can be thought of in the same way as design patterns in the field of architecture (Alexander et al., 1977) or software (Gamma et al., 2004). A dependency pattern captures knowledge about known dependencies between specific types of properties under a specific context. In the case of adding a third link to the robot, a possible pattern is to construct a rule which states that each link should have a property \textit{mass}, and include the mass of each link in the total mass calculation. Hence adding the third link should automatically update the mass calculation to include the new mass.

The word pattern should be considered as a broad term, covering different types of known relationships between design entities (whether tangible or intangible). For instance, the way a system structure relates to the structure in an analysis model (e.g. SysML to Modelica) can be seen as a pattern. The relationship between system hierarchy and the hierarchy of a mechanical assembly in a CAD model is also a pattern. Patterns exist between properties related by laws of physics, e.g. by selecting the volume of an object and the density of the material, the object’s mass can be calculated. When a CAD modeler selects dimensions of an object which lead to a known \textit{Volume}, the \textit{Mass} of the object can be found once the \textit{Material Density} is chosen. The pattern here is the known relationship between the three properties: mass, volume and density, which applies to all homogeneous physical objects.

A dependency pattern can be captured as a rule stating the type of properties covered, the type of dependency the pattern establishes, and the validity condition - stating when and under what conditions the pattern is applicable. Such a rule could be expressed as a graph grammar, as in Helms and Shea (2012); as a declarative rule, as in Hehenberger et al. (2010); or as a triple graph grammar, as in Adourian and Vangheluwe (2007). Tool interoperability is a challenge here, since properties or objects covered by a pattern could be present in a set of disparate models. Model transformations can be effectively utilized to attack the tool interoperability issue. Another challenge is how to make the hidden dependencies within models explicit, since tools may not give access to certain information needed to model a typical dependency. In Section 5.3.1, we discuss this challenge and also possible ways to formally capture a dependency pattern in the Dependency Modeling Language (DML) (whether declarative, graph based, or a TGG).

**Level 5:** Associated with the dependency pattern is another constitution referred to as the validity or applicability condition. At level 5, the applicability condition of a dependency pattern is modeled. For instance, in the case of a model transformation which executes a dependency pattern, it should be specified when the transformation rules can be applied. Modeling at this level adds further detail to the dependency model, and ensures valid application of the patterns.

As per Figure 4.1, the modeler should focus on \textit{value} when faced with the de-
cision about the level of detail at which dependency modeling is to be performed. Modeling dependencies at the higher levels provides better support to avoid inconsistencies, however at the cost of further effort. It is our future goal to determine techniques for reducing the cost of performing dependency modeling, hence increasing the value. The proposed DML and a corresponding tool that supports the DML is a step in that direction. Section 4.9 provides further details about this matter.

4.5 The services based on dependency modeling

The dependency model provides information about the known dependencies that the modeler has decided to include. Almost certainly there are additional dependencies that have been deemed unimportant or that are completely unknown to the designer (Qamar et al., 2012). At a later design phase, once some of these dependencies are known, they can be included in the dependency model. It is impractical to assume that all the dependencies are known at the design stage, although the dependency relationships of the known dependencies should be identified. Once the dependencies are captured explicitly, a number of useful services could be sought, such as checking for inconsistency or finding a suitable workflow sequence. In the following we provide a brief explanation about such services.

4.5.1 Executing dependencies

Each dependency inside the dependency network can be executed to find the property values. Before executing a dependency, it is necessary to verify that the dependencies or the dependency patterns are still valid. Remember that if a dependency is not valid, it may also lead to an inconsistency. Once the validity is ensured, then each individual dependency can be executed from the corresponding tool. For instance, in the case of the robot dependency model (Figure 4.14), the dependencies within the CAD model and the Simulink model can be executed (if still valid) to find the property values. Once the property values are updated, it is possible to check whether there is an inconsistency, e.g. in case of equality dependency binding. The dependency network provides the necessary information to perform such consistency checks, and inform the designer if an inconsistency arises.

During the execution process, it is possible that the values to some of the properties or dependencies are unknown. In this case, assumptions can be made about the valid range of property values without waiting for a particular dependency to be executed (providing these values) (Paper B). This scenario also involves making decisions about which dependencies to execute before others.

It is possible that some of the dependencies, even though known, are not included in the dependency model. For instance, there may be dependencies known to the designer but not included in the model, or it may be that one of the designers is not aware of a particular dependency. For instance, the mechanical properties of the robot are captured through a CAD model, although at a later design phase; the
mechanical properties are revisited, but this time through FEM analysis. In this case, FEM analysis introduces further dependencies to the mechanical properties, and hence should be considered together with dependencies from the CAD model. Therefore, we can suggest that among the set of known dependencies between disparate models, a viewpoint where the dependencies between any two models are looked at is not sufficient and may lead to errors. The global effect of all the dependencies has to be considered in order to understand the influence of various properties on each other. The verdict here is that completeness of the dependency network is as important as its validity. Figure 4.20 visualizes this situation by considering a viewpoint where the whole dependency network is considered for analysis, rather than parts of it.

The information gained from executing the dependency network might also lead towards suggestions for changes, both in terms of the network structure and the properties. For instance, adding a third link to the robot in the CAD model changes the structure of the network. The modeler, having authoring control over the CAD model, is the one who oversees the change in the network in such a scenario.

Figure 4.20: A viewpoint where the whole dependency network is considered. The design process begins with selecting some SPs and ends with a prediction of APs. Apart from the beginning and end of the network, APs and SPs are hidden for clarity.
4.5.2 Consistency checking

Consistency can never be fully ensured (Herzig et al., 2011). The best that can be done is to avoid as many inconsistencies as possible. As the level-of-detail of the dependency models increases, more inconsistencies can be avoided. Based on the levels-of-detail to model dependencies, modeling at level-5 is recommended (rather than level-3 since we envision that the dependency model will be supported through dependency patterns in most cases). As stated earlier, choosing which level of detail should be included in the dependency model depends upon the nature of the design problems and the size of the development organization. The aim is to focus on value, hence avoid the situation where modeling becomes too expensive. Although having some level of dependency management is imperative, further research is needed to find methods for reducing the cost. There is a strong need for a capability in the form of a tool that aids in management of dependencies. Such a tool should support modeling dependencies between disparate models, apply model transformations (using the dependency patterns), aid in evaluating consistency checks, and support in managing product variants (Qamar et al., 2012). We discuss such a tool in Section 4.9.

Consistency also requires that each stake holder has built their model conforming to the rules of the language used, conforming to the laws of nature, and according to rational preferences (Herzig et al., 2011). The same applies to modeling dependencies. The validity of a dependency and of a dependency pattern must always be checked in order to be consistent (internal consistency). As the design process moves on, the structure of the dependency network and the property values change. If the dependencies are still valid, an execution of dependencies provides new property values. It must be remembered that when the value of a property changes, consistency can only be checked if the dependencies remain the same. In other words, this suggests that after the execution of dependencies, if the results obtained are in accordance with the current knowledge of dependencies, the results are considered consistent. The network should support by highlighting inconsistencies in situations where a value of a property changes, where an addition or deletion of model elements takes place, or where a change in specifications derives a change in property values. A consistency checker can make use of the information provided by the dependency model to take control actions in such cases.

The cross in front of an SD reflects that it is possible to tear the network at that point or that a human can override the heuristics and select an SP based purely on experience or another experimental observation. As previously mentioned, in this case, the SP chosen is still considered consistent with the SD (represented by the cross on SD), even though the SP value is inconsistent with the heuristics used.

4.5.3 Change management

Managing an engineering change order is currently a difficult process, especially since it is difficult for designers to comprehend the effect the change will bring
to other functionalities of the system. Since the dependency model contains detailed information about which properties are dependent on each other, managing a change could be well supported through explicit knowledge of dependencies. For instance, if the change involves introducing a new motor to actuate a robot joint, the changes in the corresponding properties of the motor will be updated in the dependency model after the dependency execution. The dependency model then provides information about which other properties the new motor affects.

4.5.4 Human interaction

In the design of a complex mechatronic system, many stakeholders are involved. In this case, one can think that there will be numerous dependencies between models used during the design process, potentially in the hundreds or thousands, requiring substantial effort to build the dependency model. How can the value gained be maximized against the effort required to perform dependency modeling? The dependency patterns supported through model transformations are helpful in reducing human effort. The dependency modeling tool, once available, will enable the modeler to select the dependency pattern (from the library), which in turn will create the dependencies. We would like to emphasize that even though dependency patterns (supported through corresponding transformations) help in automating the process of modeling dependencies, some dependency management tasks may still require manual intervention.

The role of each stakeholder is also important in the process of dependency modeling. To gain value from the network, the dependencies need to be executed, meaning each stakeholder has to perform validity checks on the dependencies under their control, and then execute them. In addition to that, further human effort is required as the decision nodes within the dependency network are to be controlled by a human decision maker.

Another important question is how to visualize the dependency network for human understanding? This is a challenging problem due to the large number of properties and dependencies typically part of the dependency network. For instance, a human can only manage a limited amount of information at a time; hence views that expose only the part of the network relevant to the modeler are necessary. For instance, when alternative concepts are modeled, it should be possible to look at the dependencies for only one alternative, or to focus on a particular dependency within that alternative. The sequence in which the dependencies should be executed would also be managed by humans, although network analysis techniques can be used to support this activity.

We envision that a dedicated modeling language, the DML, will provide the necessary constructs and capabilities/features to make dependency modeling possible and valuable. A modeling language is described by its syntax and semantics (Harel and Rumpe, 2004). The syntax is further divided into the abstract syntax and the concrete syntax. The abstract syntax of the DML is important in order to provide sufficiently expressive constructs to model dependencies, properties and
the semantic context. On the other hand, the concrete syntax is important for human understanding. One of the future objectives of the research towards DML is to investigate the visualization possibilities for the dependency network, and to provide human interaction capabilities where actions such as selection of a node or finding properties related to a particular dependency can be entertained.

Finally, in relation to the human effort, we would like to bring to attention the role of integration frameworks. An integration framework provides a systematic tool integration approach where, in addition to providing the ability to exchange product data between multiple tools, other services such as versioning, consistency and traceability can also be utilized. Without the integration framework, tool-adapters are created considering point-to-point interactions between tools, and these are not scalable towards other tools. The integration frameworks can be effectively utilized to build and manage the dependency model. Furthermore, the framework could also enable access to other services such as version management over the dependency model. Version management of property values is important in order to trace the design evolution. It is possible that a designer makes a decision about returning to a previous design stage, for instance when analysis of a design alternative does not meet requirements, i.e. APs do not meet SPs. The version management provides the option to restore to an earlier point in the dependency network, and this service could be supported through the integration framework. We discuss the role of an integration framework further in Section 4.6.

4.6 The infrastructure for modeling dependencies

A key question to modeling dependencies is whether there should be one language where all the dependencies are captured or whether the dependencies should be represented in a distributed fashion across several modeling languages. Whilst the latter offers the benefit of providing different levels of detail, it is only viable if each language provides sufficiently expressive language constructs to create the dependency view. There are dependencies within each model (e.g. a CAD model), which are not explicitly defined as dependencies. By dependency view, we mean a view where dependencies within a model are explicitly defined. There are two ways to build such a view. The first is to provide the necessary constructs for the modeler to declare dependencies explicitly. The second is to generate this view based on reading the model and automatically extracting the dependencies. On the other hand, rather than modeling dependencies across several different modeling languages, a single language can be utilized. Here, the dependencies (within and across models) are represented in a dedicated dependency model. This single language is required to provide support for modeling dependencies at different levels of detail and at the correct level of abstraction in order to be effective. Moreover, the dependency view of each separate model needs to be available to the dependency modeling tool in order to facilitate creation of the corresponding dependencies. We believe that a single language for creating a dedicated dependency model is a more effective way
to model dependencies than utilizing current (modeling) languages to capture the same information, to avoid the problem of adding dependency modeling features to a number of languages and tools. In the following, we will focus on the challenges associated with a single Dependency Modeling Language (DML).

Since dependencies exist between properties which are captured via models built with a broad range of modeling tools, the single DML should also support modeling across different languages and tools. Hence issues such as tool interoperability and data exchange come into play. One possible solution to solve the product data exchange problem is through standards. Within the CAD community, a number of international standards were developed to make product data exchange possible. Among these standards, one of the most well-known is the ISO10303 or STEP (Pratt, 2001), which aims to describe the product data throughout its life cycle, for sharing of information between different CAD systems in a neutral file format. However, as discussed by (Gielingh, 2008), the industrial uptake of STEP and other popular standards has been poor - some key issues, such as errors in exchanged data and loss of data when using a standardized neutral model, are still not resolved. Moreover, tool-specific implementations do not conform strictly to the standards, therefore exchanging data between heterogeneous tools is often not possible without errors. As a result, organizations tend to rely on tools from a single vendor (which can be costly) or stick to documents and meetings when it comes to data exchange.

The Web Ontology Language (OWL) (W3C OWL Working Group, 2002) has been proposed for the purpose of defining vocabularies and their relationships, catering to the processing of information by computers. OWL can be used to address the model interoperability issues, such as by the UML and XML based STEP implementation (Peak et al., 2004). The dependency model can be created through generic languages such as OWL, STEP or UML. However, in order to support modeling across different languages, access and reference to model elements defined through disparate languages are required. In this regard, two solutions are possible, described as follows:

**Solution 1:** The first solution is based on an integration framework. Here, a dedicated DML and a corresponding tool are utilized. This tool will serve in the same way as other design and modeling tools, except that it will utilize the capabilities and features of an integration framework for model creation and management. The integration framework in this case helps in creating tool adapters which expose the required dependencies from each model to form the dependency view. The dependencies within each view can then be exported to the dependency model through the integration framework, as shown in Figure 4.21. In this scenario, no tool or platform takes a central position. Instead, all design activities are distributed over specific tools. It should be noted that even though tools communicate with each other directly, i.e. without going through a central platform, this approach is not to be confused with ad-hoc point-to-point tool integration. The integration platform helps in utilizing each tool’s services and data as per the needs of the design problem.

**Solution 2:** With the second solution, disparate models are transformed so
that they are available within a common platform - where the dependencies are captured - for instance, utilizing the Eclipse Modeling Framework (EMF) (Eclipse Foundation, 2009a), and transforming each disparate model to a native Eclipse representation. The DML can then be defined within Eclipse, and model transformations can be performed to create the dependency model represented in Figure 4.22. Note that, in contrast to solution 1, Eclipse takes the role of a common platform. Note also that in this solution, languages such as SysML or UML can be utilized to build a dependency model, since both of these languages are also supported in the Eclipse Modeling Framework (EMF), hence providing an effective combination.

Solution 1 is a long term solution, providing the advantages of a (needed) dedicated dependency modeling tool, and also a systematic framework for utilizing a tool’s services in a product development scenario. Solution 2 on the other hand serves well for initial testing of the DML, however it is not as scalable as solution 1 due to the limitations of using a common platform of point-to-point tool adapters. Within this thesis, we have only focused on the use of Solution 2, with the aim to migrate to Solution 1 in the future. Given Solution 2, one can ask the question “Why not model dependencies in available languages such as UML or SysML?” The following section aims to provide answers.
4.7 Modeling dependencies in SysML

In light of solution 2 discussed earlier in Section 4.6, we have considered modeling dependencies in SysML, since it is a general purpose systems modeling language which is also supported in EMF. To compare our approach to other state of the art dependency modeling methods, the Design Structure Matrix (DSM) (Eppinger and Browning, 2012) is investigated, since it is a well-known method to capture relationships or dependencies. In the following, we will explain the process of modeling dependencies in SysML, which is based on the work reported in Paper C. Section 4.8 presents a comparison between DSM and the dependency modeling approach presented in this thesis.

In the design of a mechatronic system, mechanical design, supported by Mechanical Computer Aided Design (MCAD) tools, plays a key role. Many engineering design and analysis tools are CAD-centric, and good mechatronic design cannot be achieved by performing mechanical design in isolation, rather communication and data sharing between design teams is necessary in order to manage the dependencies. Based on MCAD, several engineering analyses are performed, and if the mechanical design is changed, the input data to analysis tools has to be modified too. This necessitates managing dependencies between MCAD and other domains supporting engineering design and analysis. The other domain considered is the system design domain where a system model is created. For MCAD, Solid Edge (Siemens PLM Software, 2011) is utilized, whereas the system modeling is performed in MagicDraw (No Magic, 2011b), a UML/SysML tool. Naturally, there
are dependencies between the system hierarchy captured in a SysML model and the CAD assembly captured through the CAD model. Modeling these dependencies is the focus of this investigation. There is a small twist here, as we aim to model the dependencies explicitly in SysML, i.e. SysML is used not only to capture system hierarchy but also to model dependencies. By performing this exercise, we can gain clear insight about what is currently possible in terms of modeling that way. At the same time, the knowledge gained helps us to make decisions about the features of the DML.

In light of the solution visualized in Figure 4.22, we use an Eclipse-based tool called Cameo Workbench (No Magic, 2011a) to support the process of modeling dependencies between SysML and CAD models. Cameo Workbench provides off-the-shelf connectors which bring model data from disparate models into a common Eclipse platform (see Figure 4.23). One such connector exists for MagicDraw, which reads a native MagicDraw SysML model as an Eclipse Ecore model conforming to a UML 2.1 meta-model with a SysML profile. When it comes to CAD tools, no such connector is available, which means there was a need to construct a connector that enables data exchange with Solid Edge. This connector is written in VisualBasic.NET and utilizes the Solid Edge API (Siemens PLM Software, 2012) for communication. The connector interacts with the Solid Edge model tree and extracts the information about each assembly, corresponding parts, constraint relationships and part variables. It is also possible to parameterize a Solid Edge model and control internal model variables through a collection of parametric objects defined by the user. The Solid Edge connector reads a Solid Edge assembly and serializes it into an XML representation conforming to the Solid Edge meta-model defined in Cameo Workbench (See Figure 4.24).
SysML is a general purpose systems modeling language, but lacks the formal and detailed semantics needed to perform formal analysis (Brucker and Doser, 2007). On the other hand, a Domain Specific Language (DSL) is built based on the knowledge of concepts and definitions that domain-specific experts are used to. Therefore, combining the power of a DSL with a general purpose modeling language, such as SysML carries substantial benefits, both in terms of tool support and formal analysis (Brucker and Doser, 2007). A Profile is a light-weight extension mechanism for adding new constructs to UML or SysML. A SysML profile does not modify the underlying SysML meta-model, hence retaining the tool support. We will create a SysML profile for MCAD in order to capture the semantic details required for dependency modeling. Figure 4.25 shows the SysML profile for MCAD which will serve as the new DSL, and is based on the Solid Edge meta-model shown in Figure 4.24. Since other MCAD tools apart from Solid Edge also use similar (if
not the same) modeling concepts, a generalized MCAD profile could potentially be utilized for all these tools.

4.7.1 The process of dependency modeling between SysML and CAD

Figure 4.26 shows the process which enables creating a SysML representation of the Solid Edge model, so that dependencies between them can be modeled. The first step is to read a Solid Edge model as a native Eclipse Ecore model conforming to the Solid Edge meta-model. The next step is to transform this model into a model conforming to the SysML profile for Solid Edge. This transformation is created in the Atlas Transformation Language (ATL) (Eclipse Foundation, 2009b). A snippet of an example ATL rule set is shown in Figure 4.27, where an assembly object from the Solid Edge model is read and a cadAssembly object is created in the target model. The opposite of this ATL transformation is also implemented, where a Solid Edge model is generated as a result.

The MagicDraw connector available in Cameo Workbench reads a SysML model as conforming to the UML 2.1 meta-model with an applied SysML profile. The
Figure 4.26: The process of creating dependencies between MCAD and SysML. The dependencies are also modeled in SysML.

```plaintext
// @path SolidEdge=/ATLSysML4Cad/SolidEdge.ecore
// @path SysML4SolidEdge=/ATLSysML4Cad/sysML4CadProfile.ecore

module SolidEdge2SysML4SolidEdg;
create OUT : SysML4SolidEdge from IN : SolidEdge;

@rule cadAssembly2SysMLAssembly
{
  from src :SolidEdge!assembly
  to trg : SysML4SolidEdge!cadAssembly(
    name <- src.name,
    assemblyPart <- src.assemblyPart,
    documentAssembly <- src.documentAssembly
  )
}
```

Figure 4.27: A snippet of the ATL transformation between Solid Edge and SysML profile for Solid Edge.
Figure 4.28: A snippet of the MQL rule set, part of the transformation between SysML profile for Solid Edge and UML 2.1.

connector also extracts information about the MagicDraw environment variables, which can be captured by reading an empty SysML model through the connector, and which can be stored as a canvas in Cameo Workbench. The easy way to setup a SysML model from outside MagicDraw is then to augment the model information on top of this canvas. Therefore, the next transformation - which reads the model conforming to the SysML profile for MCAD - writes the results on top of the stored canvas. The resulting model will conform to the UML 2.1 meta-model; the SysML profile and the SysML profile for MCAD are applied to the resulting model during the transformation. In the aftermath, a Magic Draw SysML model is created conforming to the DSL. This typical transformation is written in the Model Query Language (MQL) (Sodius, 2012), a dedicated imperative model transformation language supported by Cameo Workbench. Compared to ATL, MQL provides a straightforward mechanism to implement UML profiles, and this is one of the reasons for using it. Another reason is that MQL’s imperative nature makes it easier to code transformation rules, for instance, an ‘if’ statement and the ‘for’ loop are much simpler to implement compared to a declarative language. Figure 4.28 shows an MQL rule which reads a cadPart from the model conforming to the DSL and creates a UML 2.1 stereotyped class. The opposite of this MQL transformation is also implemented.

4.7.2 Illustration: dependency modeling process for a robot

Once the transformations and the tool adapters are in place, it is possible to test the complete process through a mechatronic design example: a two degree-of-freedom robot shown in Figure 4.13. For this illustration, we will perform two steps. Firstly, we will execute the whole transformation which starts with reading the Solid Edge
4.7. MODELING DEPENDENCIES IN SYSML

model of the robot and ends with creating its SysML representation. In the second
step, we will try to model dependencies between entities of the SysML and the Solid
Edge model (created in the earlier step). The result of the first step is shown in Fig-
ure 4.29. The figure shows a robot assembly named RobotAssembly_Asm of the type
cadAssembly, which contains three parts. Part variables are shown in the proper-
ties compartment of each cadPart. The assembly constraints relationships between
parts are shown in Figure 4.30. It is now possible to perform the second step and
one way to do this is by modeling correspondence relationships as illustrated in
Figure 4.31. For instance, the lowerArm (in the SysML model) is allocated to the
link1_new.par, and the upperArm is allocated to the link2_new.par. Note that such
allocations can also be in terms of properties, for example between lowerArm and
link1_new.par. In this case, the properties can be bound to each other through a
SysML parametric diagram. This is already illustrated in Figure 4.18, which shows
how to create property bindings between system properties and analysis properties
of a motor.

Based on performing dependency modeling in SysML, it can be said that al-
though it is possible for a modeler to model dependencies there, this requires trans-
forming each model into a SysML representation, which depends upon creating
further SysML profiles, such as for MCAD, ECAD and Simulink etc. The SysML
to Modelica transformation (Paredis et al., 2010) is another example which is
standardized by Object Management Group (OMG). If dependency modeling is
performed in a dedicated DML, then the DML will provide language constructs to
capture dependencies. If an analysis model (e.g. a Modelica model) already exists,
then the dependencies can be captured between SysML and Modelica (through the
DML). If an analysis model does not exist, then the SysML to Modelica transfor-
mation can be used - firstly so that Modelica analysis does not need to be created from
scratch, and secondly to augment an analysis capability with SysML. However, the
correspondence relationships could also be captured as a dependency pattern - in
this case, the pattern would be between system structure (in SysML) and analysis
model structure (in Modelica). Therefore, we have come to the following deduc-
ton: On the one hand, model transformations are useful for attending to the semantic
overlap between models, as reflected through several examples in literature. On
the other hand, modeling and managing dependencies in an efficient manner is also
crucial to model management, which can also be supported through model trans-
formations. There is a clear difference between these two approaches; however both
seem to be necessary and we believe that the combination will equip designers with
an effective toolbox for model and dependency management.

Apart from the issue of SysML profiles, another important question is whether
SysML provides sufficiently expressive language constructs to model dependencies.
Based on our experience, we conclude that dependency modeling in SysML is weak
in terms of specifying different types of dependencies at various levels-of-detail. In
addition, it is not possible to distinguish between specified and predicted properties
in SysML. As per (Qamar et al., 2012), such distinctions are important in the
case of optimization loops, and also to avoid mistakes when the same property is
Figure 4.29: Solid Edge robot to SysML robot.
4.7. MODELING DEPENDENCIES IN SYSML

Figure 4.30: Internal block diagram showing constraint relationships between parts as was modeled in Solid Edge.

both specified and predicted. Such situations are evident from Figure 4.18 and Figure 4.31, where it was not possible to specify the type of dependency, nor the type of property, due to lack to sufficiently expressive constructs.

4.7.3 Dependency patterns

Earlier in Section 4.4.5, the notion of a dependency pattern was introduced - a pattern that gathers and illustrates known dependencies between specific types of properties under a design context. In light of this definition, the way a system hierarchy relates to a CAD assembly can also be captured as a dependency pattern. In this case, the pattern would be the relationship between the structural hierarchy in Solid Edge and SysML. Such a pattern could be supported through a transformation utilizing the association model between the Solid Edge meta-model and the SysML profile for Solid Edge (see Section 4.7.1). Similarly, we can think of other dependency patterns, such as between CAD assembly and dynamic analysis, between CAD assembly and FEM analysis and so forth.

Once several dependency patterns (and the supporting model transformations) are stored, it will be possible to realize the value of dependency modeling. For instance, the dependency model for the robot example in Figure 4.14 could be supported through multiple dependency patterns, such as between geometrical properties and inertia, and between CAD assembly and dynamic analysis. Note that such patterns may include equality dependency binding, e.g. between CAD properties and dynamic properties.
Figure 4.31: Modeling correspondence relationships between entities of SysML model and the Solid Edge model, i.e. dependencies between system hierarchy and CAD assembly.

4.8 Comparsion to the Design Structure Matrix

With regard to model dependencies and the supporting tool encompassing DML, it is natural to question how this approach compares to other state of the art approaches, one of the most well-known available in literature being the DSM. In (Eppinger and Browning, 2012), many applications of DSM are documented in various industries. To highlight the differences between our dependency modeling approach and the DSM, we consider the robot example once again. The DSM for the robot was constructed and compared against the robots dependency graph shown in Figure 4.14. Figure 4.32 shows a parameter-based DSM, highlighting the dependencies between properties of the robot. In Figure 4.32, a box contained in a cell indicates a dependency, for example between Length of Arm A and WorkSpace. The left diagonal of the DSM is highlighted in black, and all the dependencies for the robot exist below this diagonal. This means that these dependencies can be solved (mathematically) and the corresponding results found. However, any dependencies above the diagonal represent a dependency loop, meaning that the
Figure 4.32: DSM for the robot constructed in CAM (Wynn et al., 2009). For simplicity, the dependency loop as in Figure 4.16 is not modeled, hence leading to a lower-triangular DSM.
number of unknowns is larger than the number of independent equations available. Such scenarios happen when the solution requires the values of properties which are still unknown. In order to manage such cases, a reorganization of the DSM is needed to achieve a lower-triangular matrix, for example through clustering, sequencing or tearing algorithms (Qamar et al., 2012). If the loops cannot be avoided, then mathematical root finding is needed.

In the following, we will utilize the DSM and the Dependency Model (DM) of the robot example in order to highlight the similarities and dissimilarities. In terms of similarities, a parameter-based DSM visualizes dependencies between properties pretty effectively, as does the DM. Clustering or sequencing algorithms can be used through a DSM tool to find dependency groups. The same is possible with a DM, for instance dependencies could be parsed to a graph visualizing tool that provides similar layout algorithms. One such tool is Graphviz (Ellson et al., 2003) where the DOT language (Gansner et al., 2009) can be used to visualize graphs. Figure 4.33 provides a visualization of a directed graph for the robot example as presented earlier in Figure 4.14. The term Dependency Graph (DG) is analogous to the term Dependency Model (DM), the model created through the DML is called the DM, and it is possible to generate a DG based on the information available in the DM (as shown in Figure 4.33). However, it should be noted that depending upon the richness of the graph, some of the detail, such as semantic context around a property, may not be captured by a DG, and in this sense a DG is not equal to a DM.

In the case of dependency loops, a dependency will appear above the diagonal in the DSM, whereas in a DG, the loop will be visualized as in Figure 4.16. Here, for both the DSM and DG, reordering or tearing algorithms (Elmqvist and Otter, 1994) are required to tear the loop apart in order to find a solution.

In terms of differences between the DSM and the DM, it can be noted that a DSM does not differentiate between synthesis and analysis properties, nor between synthesis and analysis dependencies. In a situation, where the same property is both a synthesis property and an analysis property - for instance during optimization (Figure 4.16) - the difference between SP and AP is beneficial to avoid confusion. Similarly, by defining a dependency as an SD or an AD, there is further detail available in a DM which could be useful during decision making. For instance, the DM contains known decision points - modeled through SDs - which are controlled by a human. These decision points are important, either to break the graph, or to make a decision about dependency propagation. For example, in case of a change in specification for workspace of the robot (property WS), the DG in Figure 4.33 shows that property WS is an SP connected to $SD_1$. $SD_1$ represents a decision point where a designer makes a selection of length of each robot arm. If the amount of change in WS is small, the designer may decide not to propagate the change through the network, meaning that this change is addressed by the current values of $L_A$ and $L_B$. On the other hand, such reasoning may not be possible through a DSM. As discussed earlier, the dependencies - as modeled in a DM - can be captured at six different levels of detail, a feature which is practical since the knowledge of
dependencies evolves throughout the design process.

Another important difference between DSM and DM concerns modeling the semantic context around each property, something we argue is important for understanding the design problem. However, the semantic context is not represented in a DSM. For the DM, the semantic context is modeled through ‘part of’ and ‘value of’ relationships. Furthermore, a DSM is quite easily interpreted for small scale systems, but as the number of relationships increase, the resulting matrix may become very large - too large for humans to grasp (Qamar et al., 2012). In such a case, effective tool support is required so that the designer is provided with only the necessary information they desire. Such cases will be managed through view-points support in the Dependency Modeler. For example, to support a mechanical design view-point, the dependency model can be regenerated for visualizing only the dependencies directly related to the mechanical properties. Such a feature can be implemented since the DM has information about the model each property belongs to.

Based on the highlighted differences between DSM and DM, it can be said that

Figure 4.33: Visualization of the robot dependency graph in Dot language.
at a given instant, more information may be available through a DM than through a DSM. For the robot example, the transformation from the DSM in Figure 4.32 to a DG in Figure 4.33 is possible and could be performed during the design process. However, the opposite transformation is typically not possible since information which cannot be represented through a DSM is available inside a DM (e.g. see Figure 4.36 to Figure 4.39).

4.9 The Dependency Modeling Language

In order to convince ourselves what a dedicated tool for dependency modeling can do, we focused our efforts into building a prototype. We call this tool a *Dependency Modeler*, supported by the Dependency Modeling Language DML. In Section 4.6, two possible solutions for building the Dependency Modeler were discussed, and it was argued that the solution based on Eclipse is only short term. However, this solution does provide the tool support we need for the DML, i.e. EMF is good enough to develop and test an initial prototype. The Dependency Modeler aims to achieve the following goals:

- Understanding dependencies between different domains.
- Maintaining consistency between models and domains.
- Breaking down the design work into different types of workflows, e.g. parallel workflows for concurrent engineering.
- Automated dependency modeling through dependency patterns.

Based on our investigation on dependency modeling (Qamar et al., 2012) and the lessons learned from dependency modeling in SysML, we have come up with a set of features we deem are important to gain value from the dependency modeling. These features are:

1. Modeling of the type of property.
   
   a) If a property is chosen or selected, it is called as a Synthesis Property (SP). The classifier for the value of an SP should support specifying a property range.
   
   b) If a property is predicted through an analysis model, it is called an Analysis Property (AP). As an AP is a prediction into the future which is uncertain, it needs to be specified as a probability value. Hence, the classifier for the value of an AP should support specifying a probability density function.

2. Modeling the semantic context.
a) For a property to have an unambiguous meaning, the semantic context around each property should be specified in the dependency model. Here, we have introduced a term called Concept. As per (Thompson, 2011), a Concept is a partial specification of an artifact. Concepts are defined by putting constraints on some of the properties. With the passage of time, more constraints are put on properties, hence leading to further refined concepts. The semantic context around a property is specified through a value of relationship to a concept. Concepts are related to each other through a part of relationship, which creates the semantic context around each concept. For example, a property arm-length is a value of an arm, which in turn is a part of a robot.

3. Modeling of the type of dependency, i.e. whether a Synthesis Dependency (SD) or an Analysis Dependency (AD).

   a) An SD is a guideline or a heuristics that the designer uses to choose the value of an SP. The classifier for an SD should support referencing of the heuristics used.

   b) An AD is the analysis used to predict an AP. It can be a mathematical equation or an analysis built into a modeling tool. The classifier for an AD should support referencing to the model or equation used for the analysis.

4. Supporting different levels-of-detail in relation to modeling dependencies.

   a) If the dependency is unknown, then the classifier should allow for modeling that a dependency exists, even though the exact dependency is unknown.

   b) If the dependency is known, then the classifier should allow for modeling the type of dependency including the reference to the heuristics or the model containing it.

   c) A particular dependency is only applicable under a certain context, for instance, the applicability condition under which a model is considered valid. If the dependency is known, then a classifier should allow for modeling when the dependency is applicable.

   d) If a dependency pattern is utilized to create a known dependency, then a classifier should allow a reference to the pattern used.

   e) If a dependency pattern is utilized in creating a known dependency, then the classifier should allow modeling when the pattern is applicable.

5. Supporting different views/viewpoints.

   a) We have discussed that dependencies may be defined at different levels of detail, and that an appropriate level of abstraction for the dependency
model is to capture dependencies at the property-level. In a mechatronic design project there are many stakeholders involved with different concerns and interests. During different design stages, a stakeholder may require information about only the specific dependencies or properties. This can be attended through the provision of different viewpoints. Through selection of a viewpoint, the stakeholder provides their inputs or requirements which will be used to regenerate the dependency model. For instance, a mechanical designer could be provided with a dependency view containing only the dependencies directly related to the mechanical properties. Such views/viewpoints are beneficial in the decision making process since the modeler has to deal only with the information that fits their concerns. Based on the type of product properties captured, other views/viewpoints may be supported, for example a safety viewpoint.

6. Capturing the modeling domain.
   a) SPs and APs belong to a particular model\(^2\), i.e. a model classifier is needed which acts as a container for properties. Note that properties are related to concepts, which establish the semantic context around each property.
   b) A model belongs to a particular domain. A classifier should allow associating the domain to the model, such as CAD for mechanical design.
   c) The SDs and ADs are also parts of a particular model, i.e. a model also acts as a container to a set of dependencies.
   d) For the case of dependencies between properties of disparate models, e.g. equality property binding, the equality dependency relationship is stored within the dependency model.

7. Modeling the decision points in the dependency model.
   a) An SD reflects a decision made by a human designer. Therefore, the classifier for an SD should provide an ability to define and visualize the decision point. This will enable reviewing which points in the dependency network are controlled by the designer.

8. Completeness of the dependency network.
   a) Modeling dependencies leads to a dependency network; however, whether the network includes all the known dependencies or not is important information for the designer. If some dependencies are known (through what is already modeled) but are not included in the network, then the modeler should be reminded about it.

9. Storing the dependency patterns for reuse.

\(^2\)The word model refers to the domain-specific models, and not the dependency model itself.
a) In order to reduce the human effort in building a dependency network, dependencies that occur often could be captured and reused as a dependency pattern. For instance, the dependencies between physical properties such as geometry and material and dynamic properties such as moment of inertia. Another example of dependency pattern is between system hierarchy and CAD assembly as discussed in Section 4.7.1.

b) There should be model transformations which can execute the dependency patterns and create the known dependencies.

10. Checking for consistency.

a) In order to check for consistency, the tool should support:
   i. Execution of an individual dependency to provide updated property values. This applies provided that dependencies or potential dependency patterns are still applicable.
   ii. Once the property values are updated, consistency can be checked based on the current knowledge of dependencies included in the dependency model.
   iii. A classifier should enable raising a flag if any of the following situations occur:
       A. Dependencies are found inapplicable.
       B. The property values are found not to be consistent.
       C. Dependency patterns are found to be inapplicable.

11. Visualization of the dependency model for human-interaction.

a) The concrete syntax of the DML needs to be investigated in order to find an appropriate visualization, e.g. in terms of a dependency graph. Note that the factor of human interaction with the network has to be kept in mind. In certain cases, a human may use the dependency graph only for analysis purposes such as change management or consistency checking. In other cases, a human may investigate a typical property or a dependency within the dependency graph. Therefore, an appropriate visual representation for the dependency graph is important for understanding.

4.9.1 DML Grammar

Based on the set of features that the Dependency Modeler should support, we have developed the DML utilizing the meta-modeling support available in EMF. This is the very first prototype developed, and hence it will evolve further with time, especially since not all the features outlined above were packaged in the current specification of the DML. The set of features not implemented in the current Dependency Modeler (supported by the DML) are:

- Feature # 4: Supporting different levels-of-detail to model dependencies.
• Feature # 5: Supporting different viewpoints/views.

• Feature # 9: Storing the dependency patterns.

• Feature # 10: Consistency checker.

• Feature # 11: Visualization of the dependency graph.

Although the features not implemented are important, some of them, such as the consistency checker (feature 10), can be supported through external services. Other features are necessary for understanding (feature 5 and 11) and practicality (feature 4). Dependency patterns (feature 11) are considered as a set of rules which can be executed through a model transformation. However, further work is required to store these patterns. For the simple two degree-of-freedom robot example, it is possible to construct a suitable illustration of a dependency model, even without the availability of the missing features.

In the following, we will discuss the basic pillars of modeling language construction. This discussion is based on (Giese et al., 2007) and (Harel and Rumpe, 2004) describing basic constituents of modeling languages. The discussion will be utilized later in constructing and evaluating the DML. Models are abstractions of reality, described in some formalism or modeling language at various levels of abstraction. When constructing modeling languages, and ultimately synthesizing visual modeling environments for those languages, the following breakdown of a modeling language can be considered: the syntax (how it is represented) and semantics (what it means) (Harel and Rumpe, 2004). The syntax of a modeling language is further broken down into abstract and concrete syntax. While the abstract syntax captures the “essence” of the model, the concrete syntax is used for model visualization. For establishing syntactic correctness of a model, syntax checking is performed, and later, the meaning of a model must be understood. The meaning can be expressed through a semantic mapping function which maps every model in a language onto an element in a semantic domain. For instance, the meaning of a causal block diagram could be established through mapping onto an Ordinary Differential Equation. The semantic domain is a modeling language in its own right, and the semantic mapping is obtained by mapping the abstract syntax of a modeling language onto the abstract syntax of a semantic domain. In (Giese et al., 2007), languages were considered as (possibly infinite) sets whose sub-sets are models (see Figure 4.34).

In Figure 4.34, a set such as the meta-model contains sub-sets (represented by black dots) called models, which are models of other models. For instance, one model in the meta-models set is a model of a concrete syntax model. Similarly, a model in the transformation models set is a model of the semantic mapping function \( M \), or a model of the transformation between two abstract syntax models (\( \text{transf} \)). In the following, we begin by discussing the abstract syntax of the DML, then explaining the meaning (semantics) of a few abstract syntax classes. The concrete syntax is not explored within the work reported in this thesis; however, future work
is devised to look into this matter. The formal verification of semantics of the DML is also not reported in this thesis, and is considered as possible future work. It is understood that the DML cannot be formally specified without formal verification of a semantic mapping function. However, we believe that through the description of the abstract syntax and the meaning, we can build sufficient understanding to convince the reader about the scope of the DML. Figure 4.35 shows the abstract syntax of the DML as a class diagram. In the following, the semantics of the DML concepts are briefly discussed.

A **Concept** is a description of an artifact. A concept could refer to the ultimate product, or to its sub-components. For instance, in the case of the robot in Figure 4.13, the robot itself is a concept. A concept can contain zero to many subordinate concepts - by way of example, here are a few that can be considered for the robot: Arm1, Arm2, Controller, Motor1, Motor 2, Sensor1, and Sensor 2. All these concepts are contained under the main concept robot. Each concept is characterized by a number of **properties**, which are captured through a **model**.

A **Domain** refers to the modeling domain of a model. For example, the domain for a Solid Edge model is MCAD; the domain for a Modelica model is Dynamic
Figure 4.35: Abstract syntax (meta-model) of the current DML.
4.9. THE DEPENDENCY MODELING LANGUAGE

Analysis.

A **Model** is an abstraction of a real-world artifact (described through a *concept*). There could be many abstractions of a concept (artifact), leading to as many models built in their corresponding domains. By capturing the dependencies and their corresponding properties, each model itself contains a dependency network.

A **Property** is any descriptor of an artifact. The descriptor could be numerical, logical, stochastic or a signal. In design, two types of properties are used. Properties which are selected or chosen by the designer (Synthesis Properties) and the properties which are predicted through an analysis model, an equation or a constraint (Analysis Properties). Properties are contained inside a model, and the semantic context around a property is specified by *value of* relationships to a concept.

A **Synthesis Property (SP)** describes the value that a designer has selected for a particular property. Synthesis properties are usually defined through a range (*RangeValue*); however they could also be specified through a *Signal* varying over time. For example, a *load profile* (a signal) could be used as an SP to select the corresponding *actuator power*.

An **Analysis Property (AP)** describes the property predicted as a result of performing an analysis captured in a model, solving an equation or a constraint. APs are usually specified through a *Probability* (*ProbabilityDensityFunctionValue*), however depending upon certainty, they could be defined through a *FixedValue* or a *Signal* varying over time.

A **Property Value** is a class used to define the value of a property.

A **Dependency** defines the influence of properties onto each other. The influence could be known or unknown at a given design stage, and is described in a number of ways. The influence could be a heuristics between two or more properties (Synthesis Dependency), or a constraint, an equation or an analysis model (Analysis Dependency). The influence could also be two or more properties having equality binding between them. Dependencies are contained inside a *Model*, apart from equality dependency bindings, which are contained inside the *Dependency Model*.

There are six *levels-of-detail* in which dependencies can be defined based on situations where dependencies could be completely unknown, known without knowledge of what the dependency is, and completely known with their validity conditions as discussed in Section 4.4.5. In agreement with (Mosterman and Vangheluwe, 2004), it is emphasized that the main purpose to explicitly model dependencies is to minimize complexity. This can be achieved if:

- Modeling is done at the most appropriate level of abstraction.
- Modeling is performed using the most appropriate formalism.
- Transformations are also modeled.

Based on these proposals, the idea behind the DML is not only to provide an ability to capture dependencies at the appropriate level of abstraction, but also
to model transformations that will support in the creation of these dependencies. For this reason, two more levels-of-detail are included in dependency modeling: Level-4, where the dependency pattern and the supporting model transformations are modeled; and Level-5, describing the applicability conditions of the dependency pattern. Level-5 dependency modeling provides enough detail to know not only the dependencies, but also the transformation models that lead to explicit knowledge of the known dependencies.

A Synthesis Dependency (SD) refers to the heuristics used in selection of an SP. It is also possible that a modeler uses their experience in making this selection, and overrides the heuristics completely. An SD could have one or more SPs as its output.

An Analysis Dependency (AD) refers to the analysis (present in a model), an equation, or a constraint used to predict the value of an AP. An AD could have one or more APs as its output.

The semantics of the DML are defined in the above text. In addition to that, formal semantics of the behavior of the DML can be described by mapping to a semantic domain, for instance petri nets or finite state machines. However, this is a subject of potential future work.

4.9.2 Illustration: Dependency modeling through the DML

Once the DML meta-model shown in Figure 4.35 is defined in EMF, it is available as a language through which a dependency model can be created. We used the DML to create the dependency model for the two degree-of-freedom robot shown earlier in Figure 4.14. Figure 4.36 provides a view of the robot dependency model showing a TwoDOF_Robot concept, which further contains three concepts: Robot_Controller, RobotArm_1, and RobotArm_2. There are two abstractions to the TwoDOF_Robot, which are the two models shown as RobotMechanicalDesign and RobotControllerDesign.

Figure 4.37 shows the contents of the RobotMechanicalDesign model, under which there are three main types of entities defined: modelingDomain which is defined as MCAD, and a number of dependencies and properties. To have a detailed look at one of the dependencies, let us consider SD_1. Figure 4.38 shows that SD_1 contains two input properties, WS and PE, and two output properties Arm1_Length and Arm2_Length. It should be noted that the dependency SD_1 is modeled in exactly the same way as shown in Figure 4.14.

By looking further into the properties of SD_1 in Figure 4.38, it can be seen that each property has a defined context shown through isValueOf relationships to a concept. For instance, the property WS isValueOf TwoDOF_Robot, and property Arm1_Length isValueOf RobotArm_1. Furthermore, each property shows the dependency it is related to, through relatedDependency relationships. Lastly, each property has a defined value, such as Arm1_Length which has a range value of 50-60 cm. Note that a fixed value could instead be specified for an SP, as some
4.9. THE DEPENDENCY MODELING LANGUAGE

Figure 4.36: Dependency model of the robot developed through the DML supported through EMF.

of the specifications contain a fixed value as a potential objective. However, it is recommended to specify SPs as a range value.

One particular feature of the DML that is interesting to discuss is the possibility of equality binding between properties of two different models. To illustrate this, let us consider one more abstraction of the TwoDOF_Robot, i.e. RobotControllerDesign Simulink model as shown in Figure 4.39. Again, the model contains a number of dependencies and properties. Let us take a closer look at the dependency $AD_3$. Figure 4.40 shows that $AD_3$ contains a property $Arm1.Length$, which is a value of the concept $RobotArm_1$. This property is a part of the MCAD model $RobotMechanicalDesign$, i.e. it is contained in the MCAD model and also used in the Simulink analysis, thus a property binding. Because this property is available in the common Eclipse environment, creating this property binding is a straightforward procedure. The resulting property binding relationship is contained under the property $Arm1.Length$, and maintained inside the dependency model. Such property bindings often happen between disparate models, and hence is an important feature to be available within the DML.

4.9.3 Conclusions on the DML

Through the DML (supported in EMF), we have illustrated capturing dependencies for a robot design example in an explicit dependency model. The meta-models of Solid Edge, Matlab/Simulink, and MagicDraw SysML (as UML2.1) are available in our Eclipse implementation, and the models created in these tools can be read as Ecore models. Ultimately, model transformations can be performed to support
the execution of different dependency patterns that could exist between these three modeling domains, resulting in automatic creation of parts of a dependency model, and helping to reduce the human effort. However, manual intervention may be required anyway in situations where a modeler wishes to create certain known dependencies manually, or associates meta-data (relevant for dependency modeling) with a domain-model. As discussed, the current Dependency Modeler does not support all the features that we envision in a more complete tool. However, we believe that the current Dependency Modeler (encompassing the DML) is an important step towards a tool for modeling and managing dependencies. Further work is targeted to support the remaining features in future revisions of the Dependency Modeler.
4.10. Chapter summary

In this chapter, we have focused on the role of dependencies during the design process of mechatronic systems. We have built the theoretical foundation of the dependency modeling approach on the decision making theory proposed by Hazelrigg, and its extension in the Rational Design Theory by Thompson. The chapter proposes an approach to capture dependencies explicitly in a formal model. The tool Dependency Modeler is currently supported in EMF, and encompasses the DML which provides essential features to model dependencies between disparate models. The chapter provides examples to illustrate the dependency modeling approach, and to draw conclusions on the usefulness of the constructed dependency graph.

Figure 4.38: Contents of the dependency SD₁ of the robot.
Figure 4.39: Dependencies of the Robot controller design Simulink model.

Figure 4.40: Contents of the dependency AD₃, showing property binding to the property Arm1\_Length, which is a property of the MCAD model.
Chapter 5

Discussion

This chapter presents a short discussion on how dependency modeling would be performed in a design project, and provides an account of dependency management for hardware/software design. The future works connected to this thesis are also highlighted and the impact models are revised as per the current level of support, and the level of support expected through the future work.

5.1 Big picture

In this thesis, we have highlighted the importance of performing dependency modeling, especially within mechatronic design. Several design and analysis tools are used during the design process, satisfying the needs of a potentially large number of stakeholders. In such a diverse setting of people and development environments, communication between design teams is vital but often error-prone. A way to avoid errors is to utilize a communication strategy based on formal models, where model transformations play a key role. Within this thesis, we have illustrated an example of communication between an MCAD domain and a systems engineering domain. Here, model integration between an MCAD model and a SysML model was performed, so as to establish correspondence relationships between the SysML model artifacts and the CAD model artifacts. Another possible way of managing information transfer between domains is through integration frameworks and Domain-Specific Languages (DSLs) for tool/model integration.

While we have illustrated building model transformations between two distinct modeling languages, i.e. SysML and MCAD, it is clear that the idea of dependency modeling changes the perception of these transformations. The transformations can also be used to model dependencies between properties of SysML and MCAD models. In this sense, we have discussed how the MCAD-SysML transformation can be utilized to support a dependency pattern between system hierarchy and the CAD assembly. There can potentially be many other recurring dependency patterns which can be supported through such transformations. Note that our aim
is not point-to-point dependency modeling, nor point-to-point integration between
domains. Rather, we aim for modeling and managing dependencies between various
design and analysis models (used within different domains), so as to support a
holistic approach where dependencies between many design and analysis models
play a key role in the formulation of an overall design study. It is in this context that
the designer could ultimately achieve the overall value that dependency modeling
brings.

The dependency model will be built as soon as information about a product
design is captured in models. The product design process also includes a synthesis
phase, where the whole design space is searched to formulate design concepts that
fulfill the design requirements. Model transformation could also be used to search
for solutions within the design space during the synthesis phase. Some of the related
work in this area is reported in (Helms and Shea, 2012) and (Kerzhner, 2012). For
dependency modeling, the initial synthesis phase has not been considered, mainly
because dependency modeling is to be performed after the candidate solutions have
been found through the synthesis phase. Once the candidate solutions have been
formulated, dependency modeling will support their evaluation to identify the best.
The robot design case study (Figure 4.13) considered one design concept that came
out of a synthesis phase. As soon as this design concept was modeled and analyzed,
dependencies were captured inside a dependency model (A hospital bed design
synthesis phase has been described in Paper D).

The proposed meta-model for the DML is based on the core terminology for
modeling dependencies as reported in Paper B. Similarities can be seen between
the dependency modeling approach and the DSM, and one might be tempted to ask: why not introduce the core concepts we have proposed into the DSM? While
we do argue that the core concepts on dependency modeling are not available in
DSM, the DSM is still quite powerful and easy to use for small scale systems.
However, when it comes to human-interaction with the dependency network, we
believe that the matrix-based visualization of a DSM for a complex system is not so
well-suited. There could of course be synergies sought between the two approaches
where DSM is utilized in the beginning and the DML is utilized later to create
a dependency model. This could be supported through transformations between
DSM and Dependency Graph, as reported in Section 4.8.

Software/Hardware design also plays a major role during the design of mecha-
tronics. Co-design of hardware and software has been studied by many researchers,
as reflected through the recent advancements in Mode-Driven Engineering. For
instance, for a mechatronic system, control design can be performed in Matlab/
Simulink, which also supports code generation. However, in terms of performance
of the control system when running on an embedded computer, hardware character-
istics have to be taken into account. Therefore properties such as Worst Case
Execution Time (WCET), Delay, Priority, Processing Power, and Clock Cycle are
vital considerations within the SW/HW design area. Furthermore, it is possible
that software design is governed by design rules specifying which dependencies are
allowed. Such rules are formulated to make sure that there are no unintended depen-
dependencies between tasks performed by the software (Sangal et al., 2005). In addition to that, there may also be rules stating what type of dependencies are required to be managed between hardware and software design teams. The work reported in (Törngren et al., 2012b) is an example of managing dependencies through design contracts between controller and software design. Therefore, although we have not discussed the dependency modeling approach in relation to software/hardware design, the approach carries potential benefits in terms of model and dependency management for those domains.

Based on the prescriptive work on the Dependency Modeler supporting the DML, and on model integration supporting communication between models, the impact model can be revised. Figure 5.1 shows how the developed support affects different factors. Firstly, consistency checking has not yet been implemented, although there is a great deal of evidence that dependency modeling will support consistency management. Hence, no effects of the developed support are shown on the level of consistency. Moreover, the time to develop can still not be verified. Such verification would require performing a case study, preferably in an industrial setting, and this will be targeted by future work. Despite this, the developed support helps to avoid design faults, hence increasing the reliability of the system (measurable success factor). Dependency modeling also provides support when managing an engineering change by tracing the change to the corresponding sections in the dependency network. Finally, by performing dependency modeling of alternative concepts, comparison between concepts can be made quite effectively. Therefore, the initial reflection is that the developed support satisfies the success factor, and further evidence can be obtained about this through a future case study.

5.2 Abstractions between dependency models

In relation to modeling dependencies explicitly, capturing dependencies between properties was considered an appropriate abstraction. However, it is possible to raise the abstraction level and think of dependencies between Files, where a file can contain one or many Models, and a Model is the container of several properties and dependencies. The following discussion aims to elaborate on such abstractions.

Figure 5.2 shows a rather abstract view illustrating how the Mechanical Design and Controller Design views of the Robot are dependent on each other. For instance, there are dependencies between an MCAD model within the Mechanical Design view and a Simulink model in the Controller Design view. Originally, the dependencies between these two models are not explicit, but by using the dependency modeling approach, the dependencies within and between the models can be made explicit, as illustrated in Figure 5.3. Later, these dependencies are represented in a dependency graph as shown in Figure 5.4.

In connection to human interaction with the dependency modeling, it has to be kept in mind that both the MCAD model and the Simulink model are built and managed by a human decision maker/modeler. It may happen that both modelers
Figure 5.1: Impact model revised after the prescriptive study 2.

Figure 5.2: An abstract representation of dependencies between two views of the robot.
Figure 5.3: Making dependencies explicit within and in between the mechanical design and controller design views.
wish to visualize the dependencies in between their models with as little detail as possible. Figure 5.5 illustrates a less-detailed view of the dependency model previously shown in Figure 5.4. Note that now only the properties which play a role in dependencies between the two models are shown, and the rest of the details are hidden. This makes it simpler for the two modelers to interact and discuss the dependencies between their models. In Section 4.9, we discussed the availability of multiple viewpoints/views with the DML, where a modeler can regenerate a view of the dependency model based on the properties or the dependencies they are interested in. This feature can also be used for visualization as illustrated in Figure 5.5, where the dependency model is recreated based on the needs of both MCAD and Simulink modelers.

In relation to customizing the dependency graph based on the needs and preferences of various stakeholders, a further abstraction for dependency modeling can be considered - in this case, in relation to the conceptual space of a theory (within a domain), which forms the basis of various properties. Figure 5.6 illustrates this,
Figure 5.5: A less detailed view of the dependencies between mechanical design and controller design for a robot.

where dependencies are modeled as relationships between different types of theories - for instance, the theories of kinematics, geometry, dynamics and control design. All theories have their own conceptual space, defining the relevant essential concepts (Yoshikawa et al., 1994; Tomiyama et al., 1989). Utilizing these concepts, a model can be built conforming to the conceptual space of the theory. Considering the dependencies between dynamics and control design (in Figure 5.6), such an illustration, even though abstract, brings value to the modelers in different situations - for instance, while considering what types of dependencies may exist between different views before actual modeling is performed, a problem relating to the structuring of views (ISO/IEC/IEEE 42010, 2011). Figure 5.6 also illustrates the idea of dependency patterns (see Section 4.7.3), using the dependencies between dynamics and control design. Keep in mind that there may be more than one pattern that could be captured between these domains, e.g. a pattern between controller gains and inertia is different from a pattern between inertia and resonance frequency. Such distinct patterns - once formally defined - could be reused in different projects. Through dependency patterns, one can think of dependencies - not between property instances but rather between types of properties - that could be applicable in several instances. Hence, the dependency modeling approach does not only cater for dependencies between property instances, but also enables modeling of the patterns that generate those dependencies. These concepts help generalize the applicability of the dependency modeling approach towards many domains.

The illustration in Figure 5.6 also relates closely to the work on Design Contracts (Derler et al., 2013; Törngren et al., 2012b), where there could be one or more contracts specified between dynamics and control design. Such contracts represent the rights and obligations of each party (Derler et al., 2013), which in this case are the control designer and the modeler performing dynamic analysis. Both parties have to ensure that they fulfill the agreement during the design process in order to achieve harmony. A possible future investigation could be to transform design contracts into dependency graphs and vice versa. In this way, one could think of
modeling dependencies at different levels-of-abstraction. For example, Figure 5.3, Figure 5.5 and Figure 5.6 illustrate dependency modeling from a lower abstraction level to a higher abstraction level. These ideas could be extended towards the multiple views that can be created for a product, for example Figure 5.7 shows many views that can be considered for the robot example.

5.3 Future work

In the scope of the work reported in this thesis, different directions of future work have been sought. This includes further work on the development of the DML, on the use of an integration framework with dependency modeling, and on a case study in an industrial setting to assess the efficiency and effectiveness of the dependency modeling approach.
5.3. FUTURE WORK

5.3.1 Evolution of DML

Within this thesis, the abstract syntax of the DML and the meaning of corresponding terms have been explained together with the Dependency Modeler supported through the EMF. However, as highlighted in Section 4.9.1, a few of the features of the DML are still not available in the current implementation (namely those for modeling dependencies at different levels-of-detail, supporting different views/viewpoints on the dependency model, implementing a consistency checker, and storing the dependency patterns). In addition, the concrete syntax of DML - which is important for visualization of the dependency graph - has not been thoroughly investigated, and we have instead relied on block-based visualization (see Section 4.9.2). Also, formal verification of the semantics of the DML is not covered, but is considered as a possible future work. Even so, we can illustrate the value of the current DML through a robot example even without these features, but they are clearly vital for maturity and completeness of the support we seek. In the following, a short discussion on each feature is provided to highlight the potential of the different aspects of the future work.

Modeling dependencies at different levels-of-detail

Supporting dependency modeling at different levels-of-detail is vital for practical reasons, since knowledge about dependencies increases as the design process evolves, and not all the dependencies are known in the beginning. In the current implementation of the DML, dependencies are considered either known (represented in the

Figure 5.7: Multiple views on the robot.
dependency model) or unknown. In future, we aim to extend this to allow modeling that a dependency exists without actually knowing what it is. Future work on capturing and using dependency patterns will provide two more levels-of-detail.

**Supporting different views/viewpoints**

In order to tackle the complexity which may arise from the large amount of information typically present in the dependency model, different viewpoints/views may be necessary. Such a feature will enable the modeler to construct a dependency view based on their requirements, hence only dealing with a limited set of information. For instance, a mechanical designer may require a dependency view showing only the dependencies that influence mechanical properties. Such a view can be generated by searching the dependency model for dependencies directly related to the mechanical properties (e.g. properties which are a part of the mechanical design model). Ultimately, viewpoints/views carry a high utility in terms of human interaction with the dependency network.

**Dependency patterns**

When modeling dependencies, one question that comes to mind is how to make the dependencies explicit from models that only contain implicit information. In reality, some design tools may make it difficult to access certain information, or worse, do not make it available at all. Although we acknowledge the challenge here, we believe that capturing dependency patterns is the best way to extract information from models developed through such tools. The aim is to capture known relationships between different entities as patterns. Some of these relationships are known through laws of physics, e.g. \( \text{Mass} = \text{Density} \times \text{Volume} \), \( \text{Force} = \text{Mass} \times \text{Acceleration} \). Such laws are utilized in analysis tools but may not be available explicitly, so capturing them as a dependency pattern will help in making dependencies (covered by various laws of nature) explicit. In order to illustrate this, consider Figure 5.8, where two graphs are shown. The left hand side visualizes a CAD model, where a Part contains the following Variables: Volume, Material Density and Inertia. Once the modeler makes a selection (SPs) of Part Geometry (and as a consequence it’s Volume) and Material Density, the analysis built into the CAD tool predicts (AP) the corresponding Mass (or Inertia). The CAD tools may not make the analysis explicitly available for the corresponding dependency to be explicitly modeled. However, if captured as a pattern, the law \( \text{Mass} = \text{Density} \times \text{Volume} \) already provides information about the dependency - which includes information about both the dependency relationship and the type of dependency (in this case an AD). The graph on the right hand side shows the explicit dependency between the same properties (as on the left hand side). Thus by considering the left side and the right side of Figure 5.8, we can think of a dependency pattern captured as a graph grammar.
5.3. FUTURE WORK

One cannot assume that all the patterns are known through laws of nature. There can be other types of dependency patterns, such as between system structure and CAD assembly hierarchy. This pattern was discussed earlier in Section 4.7.1, and is again reflected in Figure 5.9, showing the relationships between Solid Edge meta-model and SysML4SolidEdge profile. Other examples are patterns between system structure and analysis model structure (e.g. SysML to Modelica), and between CAD Assembly and Dynamic Analysis (e.g. Solid Edge to Modelica). Many such patterns have already been implemented in literature, with an attempt to manage the semantic overlap between two models. For example, the SysML to Modelica transformation is reported in (Paredis et al., 2010; Reichwein et al., 2012), whereas Adourian and Vangheluwe (2007) discuss the Solid Edge to Modelica transformation based on TGG. By treating such associations and their supporting transformations as dependency patterns, our aim to make dependencies explicit is achievable. To summarize, we can confidently declare that dependency patterns are vital - both to reduce the human effort required in creating the dependency models, and also to make the overall dependency modeling approach work - and hence future research into this topic is likely to carry a significant benefit.

Consistency Checking

The dependency model contains useful information for implementing consistency checking. Many of the approaches currently available in literature focus on rule-based consistency checking. Considering these approaches, when an inconsistency is spotted through the dependency model, then a corresponding consistency rule has to be evaluated in order to decide on the appropriate action. The challenge is that it may not be possible to capture all the inconsistency scenarios in terms of rules. On the other hand, some inconsistencies may still be allowed (in some cases), hence requiring no action. However, implementing a consistency checker is
Figure 5.9: Association between Solid Edge meta-model and SysML4SolidEdge profile.

a research topic in itself, and is not yet tackled within this thesis. Nevertheless, we believe that the dependency model makes it easier to perform consistency checks. If a consistency checking tool is available, its services can also be utilized through an integration framework as illustrated previously in Figure 4.21.

Concrete syntax of the DML

In terms of the maturity of the DML, the concrete syntax is not thoroughly investigated, and a block-based visualization available in EMF is used. Although such visualizations (e.g. Figure 4.36 to Figure 4.40 - shown earlier in Chapter 4) provide basic representation of the dependency graph, the block-based visualization may not be the best. Also, layout capabilities to reorder the dependency graph under
clustering and sequencing are not currently available in the Dependency Modeler. However, using the DOT language, we have illustrated layout possibilities, shown in Figure 5.4.

**Formal specification of DML semantics**

Revisiting Figure 4.34, for a language (such as DML) a model in the Abstract Syntax set needs a unique and precise meaning, which is done by providing a mapping to a Semantic Domain. The resulting semantic mapping function specifies the meaning by mapping every model in a language into an element in a semantic domain (Giese et al., 2007). The mapping can be formally captured as model transformations, e.g. through Graph Grammars, which can capture complex behavior through analyzable and executable graphical rules. For the DML, we aim to define a mapping to Petri-nets, and verify the DML grammar through analysis such as reachability testing. It is also possible to check the resulting semantic domain model for deadlock. For example, in Figure 5.4, the property \textit{Sensor1\_Resolution} has no transition going out - in other words, if a change in \textit{Sensor1\_Resolution} occurs, the current model does not provide the means to manage such a change.

It is important to note that although the meaning of a model can be specified through the semantic domain mapping, the verification of semantics is based on the correctness of the mapping function, which guarantees correctness of models created through the semantic domain language. However, performing an exhaustive verification may not be possible - as this requires searching the whole design space - where there may be an instance model (of the semantic domain language) that challenges the validity of the mapping. Formal verification of a language is a challenging subject - a fact proven for languages such as UML and SysML, where semantic variations can lead to inconsistencies (Chanda et al., 2010; Broy et al., 2006).

**Human effort**

In terms of the human effort required to build and manage dependencies through the dependency model, consider the following situations:

1. A modeler, having knowledge about certain dependencies, creates them manually inside the dependency model.

2. A domain-specific modeler (while modeling) designates a certain set of information as relevant for the dependency model. For example, in Solid Edge, one can populate a table of parameters which are externally accessible and which can be used to control the native Solid Edge model.

3. A modeler utilizes an available dependency pattern (within the Dependency Modeler) to create a portion of the dependency graph automatically.
We have discussed that different types of patterns are known from theory, heuristics or practice, but making them available in the Dependency Modeler requires an effort. All things considered, it may not be possible to fully automate the process of building a dependency model, and human intervention is likely to be required in one way or another (e.g. situations 1 & 2 above). Within the time frame of this project, a study that enables predicting average human effort could not be performed. Such a study could be done qualitatively or quantitatively, and is an objective of future work to predict the average human effort spent in building and maintaining dependency models.

5.3.2 Role of an integration framework

In Section 4.6, two solutions to model dependencies were outlined. The second solution was based on EMF, which is what we followed in this thesis. The first solution is more long-term, and utilizes an integration framework to enable information exchange between disparate models. iFEST is one such framework. The idea is that no tool takes a central position in the framework; rather each tool is a provider or a consumer of services and data to/from other tools. By defining each tool’s services and data per the pre-determined specifications, tools can communicate directly with each other at run-time. The iFEST framework utilizes a common web-based technological space as advocated by the OSLC standard (OSLC Core Specification Workgroup, 2010) and Service Oriented Architectures (SOA) (Erl, 2008). In this case, each tool’s interface needs to map from its specific technology to the common technological space in order to expose the requested and provided services to other integrated tools. Revisiting the first solution as shown again in Figure 5.10, a tool interface could be seen against every tool. A tool interface implements a standardized specification of a tool’s services. For example, the interfaces of all CAD tools need to conform to the same CAD tool-type specification. In this sense, the Dependency Modeler will utilize a tool interface (from each tool) that provides explicit information about dependencies as per the DML specifications. Languages supporting model and tool integration are helpful for generating such interfaces automatically, e.g. Simko et al. (2012) and Biehl et al. (2012). As can be seen in Figure 5.10, several other services could also be made available through such an approach, for instance version management and consistency checking.

5.3.3 Case study in an industrial setting

Once all the features of the DML are available, and an initial implementation through the integration framework is performed, the next step is to test the approach through an industrial case study. In such a case study, the design tools specifications would be laid down in collaboration with an industrial partner. The required support capabilities for the study would also be stated, e.g. checking consistency, or performing optimization. The dependency modeling would then be performed using the prototype development environment of the Dependency Mod-
5.3. FUTURE WORK

The efficiency and effectiveness of the Dependency Modeler in supporting the decision making process would be monitored in terms of value.

5.3.4 Relating Dependency Modeling and Design Contracts

Earlier in Section 3.4, we introduced the contract-based design approach, and went on to discuss its possible connection to dependency modeling in Section 5.2. Thinking about the big picture of a mechatronic design process, the involvement of multiple disciplines leads to new problems, which can be seen from at least four different perspectives:

1. **People:** The problem of communication between stake-holders having a variety of backgrounds and concerns.

2. **Models:** The partitioning of the overall design problem into domain-specific ones, and the use of a number of appropriate formalisms to support the modeling process. As a result, issues such as dependency and consistency management will need to be addressed.

3. **Tools:** The models developed in step 2 are supported by a wide variety of tools, which themselves require integration in order to support design integration. Issues such as data exchange standards, version management and tool-integration are important here.
4. Process: Mechatronic design is essentially a multi-disciplinary process, where the dispersion of product information among people, models, methods and tools has to be taken into account.

In light of the above taxonomy, and keeping both the dependency modeling and design contracts approaches in mind, it can be said that the former approach aims to provide support for managing disparate models (Models perspective), whilst design contracts can be thought of as an agreement between people (People perspective), e.g. between mechanical engineer and control engineer. In this regard, integrating the two approaches is seen as a potential benefit, where we can envision that the properties considered in a design contract could be related to the properties considered in the dependency model. Therefore, it is our aim to study the possible relationships between these two approaches, and define possible transformations between them which can be supported through tooling (Tools perspective).

5.3.5 Studying the relationships between Design Process and Dependency Modeling

Thinking of the engineering design process and Hazelrigg’s decision theory, one can visualize design as a process where predictions about the required characteristics of the product are made. With reference to the robot example used in this thesis, one could think of making the following predictions:

- Weight
- Work space
- Time for End-to-End movement
- Positioning accuracy
- Cost

To make the above predictions for the robot, in Figure 5.11, one possible (hypothetical) design process flow is represented, beginning with mechanical design, and followed by electronic and controller design. The final steps are to optimize the whole system and then perform cost analysis. By following this design process, the design team can make predictions about the product characteristics at the different time instances shown in Figure 5.12.

As discussed previously, one possible service arising from dependency modeling is to decide the workflow process. Based on the information about the design process, it is possible to visualize the resulting dependency graph once the design activities take place (see Figure 5.13). By studying Figure 5.11, Figure 5.12 and Figure 5.13, one can find relationships between the activities in the design process and the corresponding paths in the dependency graph. It is the goal of future work to study these relationships so that dependency graphs can be created once the design process is described.
5.3. FUTURE WORK

Mechanical Design
Electronic Design
Controller Design
Optimization

Time (months)

Cost Analysis

0.5 1 1.5 2

Figure 5.11: Design process against time.

Workspace  Weight  Positioning accuracy  E-E time  Cost

0.5 0.75 1.25 1.75 2

Time (months)

Figure 5.12: Predictions against time.
Figure 5.13: A hypothetical dependency graph for the design process of Figure 5.11.

5.4 Chapter summary

This chapter summarized the work performed in this thesis and represents it in a revised impact model showing the influence of the developed DML, the Dependency Modeler and the model integration approach on the success factors. Furthermore, the chapter highlighted proposed future work connected to this thesis. The future work is further divided into three parts: the first involves further development of the DML - from incorporating the remaining features to formal verification of semantics; the second involves constructing a working example of modeling dependencies through the Dependency Modeler supported by an integration framework, hence also testing consistency checking and version management services; the third includes performing a case study in an industrial setting, to establish the value gained through dependency modeling, along with studying its effectiveness at detecting errors during design (measurable success factor). In the methodology presented in Chapter 2, the future work is included as a potential descriptive study 3, evaluating the developed support. Figure 5.14 shows the expected influences of the future work, in which the integration framework provides access to services - for managing versions and for detecting and avoiding inconsistencies. Note that it is expected that modeling dependencies may require further effort from the modelers. Hence a reduction in development time may or may not be seen and cannot currently be predicted, since design time could be increased if the dependency patterns are not available and the modelers have to perform manual work as a consequence.
Figure 5.14: Impact model showing the influences of the future work.
Chapter 6

Conclusions

A significant number of challenges are faced during the design of mechatronic products. This chapter discusses the contribution and benefits of the dependency modeling approach reported in this thesis towards addressing some of the key challenges in mechatronic design. The chapter also provides an account of the research questions and the hypotheses addressing the research questions, to provide a verdict on which of them are addressed through the work performed in this thesis.

6.1 Relationships between appended papers

This thesis is based on the six appended papers, written during different phases of the work. Paper A provided a comprehensive account of mechatronic design challenges, and helped in highlighting the most important challenges relevant for the work performed. This led towards two main directions for the research work (Figure 6.1), the first including the work on dependency modeling, and the second including the work on model integration and investigations into the common language. Paper E provides initial ideas which were very useful in developing the dependency modeling approach presented in Paper B. On the other hand, Paper F proposed an initial idea of integrating models through a common system model. This idea flourished further, investigating the integration between system-level design and domain-specific design. The common language used in creating the system model was also investigated, and integration based on a common product model was sought, as presented in Paper D. At this point, the work took a detour, and instead of focusing on integration through a common system model, dependency modeling was considered as a more favorable approach. Paper C describes these findings, where model integration between a system model and an MCAD model is explained, but in the context of supporting the dependency modeling. Figure 6.1 provides an illustration of this evolution, showing how the different papers relate to each other.

In the following, we will revisit the developed support, measuring it against the
research questions formulated earlier in Section 1.5, in order to figure out how well the work performed addresses the research questions.

### 6.2 Communication between mechatronic domains

The communication problem is related to the following research question:

\textit{RQ4: How to support communication e.g. in terms of information exchange between design domains in a mechatronic design context?}

To answer this research question, the following hypothesis was proposed:

\textit{Hyp4: To facilitate information exchange between disparate tools, a model based strategy supported by tool adapters and model transformations helps in providing the communication support.}
Within the topic of model transformations to support communication between domains, two different types of strategies have been investigated. The first one was based on using a common product model to support communication between domain-specific models. The second one was based on capturing dependencies between domain-specific models through a dependency model. For the first strategy, where SysML was considered as a common language, it was argued that modeling dependencies requires expressive language constructs (which are currently not available) and building many domain-specific SysML profiles. Even then, the dependencies cannot be captured or managed explicitly through such a solution. The second strategy, which proposes capturing dependencies explicitly in a dependency model, was considered far more effective. The role of transformations here is to automate the creation of the dependency model through dependency patterns. Within the thesis, the dependency modeling tool support was provided by constructing the DML through EMF. An illustration of model transformation supporting a dependency pattern between system hierarchy and CAD assembly was made. The work on utilizing model transformation verifies hypothesis 4. A refinement of hypothesis 4 was considered in hypothesis 7, which is as follows:

**Hyp7:** A tool integration framework helps in creating tool adapters in a systematic fashion, and provides a sustainable infrastructure to support communication between disparate tools.

We have suggested that in the future, an integration framework utilizing the Dependency Modeler will provide a more scalable solution to the communication problem. However, further work is required in order to gain evidence to verify hypothesis 7.

### 6.3 Dependency modeling

The thesis provided a comprehensive study to figure out the most important challenges in mechatronic design. It was discussed that lack of dependency management is one of the main factors behind many product related challenges. Several of the research questions are related to dependency management, revisited here one by one.

**RQ1:** What kind of dependencies exist between different domains in a mechatronic design context?

For this research question, the nature of dependencies was studied through descriptive literature, and through a robot design case study. The following hypothesis was considered as an answer to RQ1:

**Hyp1:** By investigating dependencies between design domains through different case
studies, it is possible to find out the nature of dependencies and the requirements to manage them.

Based on the work reported in Paper B and Paper C, it can be said that an adequate level of abstraction for the Dependency Model is to capture dependencies at the level of properties (captured through disparate models). Moreover, through the robot case study, the nature of dependencies and the requirements to manage these dependencies were found. Therefore, hypothesis 1 is verified. The second research question concerning dependency modeling was:

**RQ2:** What type of approaches can be used to manage dependencies while performing design activities in different domains?

A detailed study was performed to answer this, and is reported in Paper B. The question was further broken down into four refinements:

- **RQ7:** How detailed should the dependency representation be?
- **RQ8:** How can dependency models be managed, and how will humans utilize these models?
- **RQ9:** What type of support is available in current modeling languages to model dependencies explicitly?
- **RQ10:** How can the value gained be quantitatively compared against the effort required in creating and managing dependency models?

In order to answer these research questions, the main hypothesis was:

**Hyp2:** An approach that makes dependencies explicit by capturing them through a formal model provides a foundation to manage dependencies.

The main idea was to formally capture the dependencies in a model. Six levels-of-detail were considered to capture these dependencies. The support provided by current approaches to capture dependencies was considered, and it was concluded that the level of support in current languages was not adequate. Paper B proposed the terminology for a new DML, which was later built and illustrated through a robot example in Section 4.9.2. Based on this work, it can be concluded that dependency modeling is valuable and hypothesis 2 is verified. However, the answers to RQ8 and RQ10 are not available completely, since this requires further work on the visualization of dependency networks, and a case study where the value gained through dependency modeling is further measured.

Overall, it can be said that modeling dependencies explicitly is clearly beyond the current state of the art and practice. The approach proposed in this thesis is founded on the theoretical basis of decision-based design. By considering the
6.3. DEPENDENCY MODELING

synthesis and analysis nature of both the properties and the dependencies, we argue that the chances of designer error are reduced.

The thesis presented a new language for modeling dependencies called DML, and provided a discussion on DMLs abstract syntax, semantics and possible model transformations that play a vital role in terms of user experience. The thesis also presented possible further work on DML, such as on concrete syntax and formal verification of semantics. The dependency modeling approach encompassing the tool support and the DML are the main scientific contributions of the research reported in this thesis. However, even with the advantages, we believe that our solutions also introduce their own complexity into the design process. Atkinson and Kühne (2007) call this *accidental complexity*, which is not inherent to the design problem but arises due to the methods and solutions used to solve it. In terms of the tool Dependency Modeler, accidental complexity exists when capturing dependency patterns and during human-interaction with the dependency graph, and our aim is to reduce this complexity and make the whole approach valuable.

6.3.1 Consistency checking

Several services can be built based on dependency modeling, one of the most important ones being to avoid inconsistencies. Although consistency can never be fully guaranteed, we argue that the knowledge available in the dependency model provides substantial information to avoid many *internal inconsistencies*, for instance between property values. The main research question and hypothesis related to consistency management were:

*RQ3: Given that there are dependencies between disparate models, how can consistency be ensured?*

*Hyp3: Making dependencies explicit in a dependency model provides the necessary information to facilitate consistency management.*

Based on the work reported in Paper B, Paper C, and Chapter 4 of this thesis, it can be said that there is strong evidence that dependency modeling facilitates in consistency management. However, a consistency checker (with the corresponding consistency management rules) has to be implemented to validate this argument, which is a part of the future work. Therefore, hypothesis 3 has a strong chance of being verified in the near future.

6.3.2 Assessing consequences

Another benefit that dependency modeling brings with it is comparing alternative design concepts. This is mainly related to the following research question and its hypothesis:
RQ6: During design, a number of alternative concepts may be considered. How can the consequences of choosing between these alternatives be assessed?

Hyp6: Dependency modeling supports assessment of consequences for alternative concepts.

Once the alternative concepts are modeled, the dependency model can be built. In accordance with the Decision Theory (see Section 4.1), to construct an overall value function, properties from disparate models are coupled into a design study, where the selection criteria can be formulated based on the designer’s preferences. The dependency model provides the necessary details to build such a design study as the coupling between design models is already captured here. Therefore, it can be said that there is a strong capability in dependency modeling to support assessment of consequences of alternative design concepts. The assessment can also be performed when considering a change management scenario for one or more of the alternative concepts. In this case, the dependency model provides information about where the change will happen, and what will be the corresponding change, provided that dependency modeling was performed at adequate level-of-detail. Therefore, hypothesis 5, although not completely verified at the moment, has a strong chance of being verified through further work.

6.3.3 Optimization

One of the important issues when designing mechatronic products is regarding optimization. As documented by many studies, due to the cross-disciplinary nature of mechatronics, performing a design optimization is either too complicated, or too time-consuming. The main research question tackling the optimization issue was:

RQ5: Given that there are dependencies between disparate models, what is a suitable method for performing design optimization?

To search for a suitable method, the following hypothesis was considered as an answer to the research question.

Hyp5: Dependency models provide adequate details in order to support a design optimization where disparate models are lumped into an overall design study.

One of the obstacles to performing a design optimization study in mechatronics is that it requires information exchange between disparate models, many of which are difficult to couple with each other due to the obstacles faced in data exchange. Section 3.3 and Section 4.6 provided a discussion on challenges in product data exchange. This problem partly relates to the problem of assessing consequences, requiring coupling between disparate models in a design study governed by an optimization algorithm. Performed at an adequate level-of-detail, depen-
dency modeling captures the coupling between disparate models. A possible future work is to study the possibilities of supporting design optimization through a dependency model (where the necessary dependencies relating to the optimization function are captured). Therefore, hypothesis 5 again has a strong potential to be verified, although it is not fully verified at the moment. There are research approaches targeting early optimization of mechatronic products without utilizing detailed models. Malmquist et al. (2012) and Kerzhner (2012) are examples of the work in this direction. Nevertheless, optimization is an important research issue for mechatronic product development, and further efforts are essential in order to completely explore the potential of dependency modeling in this area.

6.4 Chapter summary

This chapter highlighted the main contributions of the work reported in this thesis, and measured it against the research questions and hypothesis formulated for this work. The chapter provided a short discussion on the hypotheses verified and those which will require further work. The dependency modeling approach reported in this thesis carries a significant potential to address some of the key mechatronic challenges. The chapter also signifies the scientific contribution of the thesis work, and weighs up the benefits that can be gained through the proposed future work.
Chapter 7

Summary of Appended Papers

This chapter provides a brief summary of the appended paper on which this thesis is based. The papers are focused towards two main issues: dependency management, and model transformations to support communication between mechatronic domains.

7.1 Paper A

Challenges in Designing Mechatronic Systems

This paper aims to identify the major challenges in mechatronic design by conducting a systematic and thorough survey of the most relevant research work. Solutions proposed in literature are assessed and illustrated through a case study in order to investigate if the challenges can be handled appropriately by the methods, tools, and mind-sets suggested by the mechatronic community. Using a real-world mechatronics case, the paper identifies the areas where further research is required, by showing a clear connection between the actual problems faced during the design task and the nature of the solutions currently available.

7.2 Paper B

Dependency Modeling and Model Management in Mechatronic Design

The partitioning of the overall design problem into domain-specific problems leads to dependencies between product properties, properties which can be captured through disparate models. This paper investigates the nature of dependencies and proposes appropriate terminology, taking into account the synthesis and analysis nature of both the properties and the dependencies. This terminology will be the core of the new dependency modeling language (DML). The concepts related to dependency modeling are then illustrated through a simple robot design example,
where the creation and importance of a dependency model are explained. Practical approaches for consistency management and model management in the presence of dependencies are also studied. Six levels-of-detail in modeling dependencies are presented, emphasizing that modeling at a higher level-of-detail ensures that more inconsistencies are avoided.

### 7.3 Paper C

*Managing Dependencies in Mechatronic Design: A Case Study on Dependency Management between Mechanical Design and System Design*

The paper highlights dependencies between two important modeling domains in mechatronics: system modeling and Mechanical Computer Aided Design (MCAD). Mechanical design (supported by CAD tools) establishes a backbone for the overall design concept, hence dependencies relating to the mechanical design cannot be overlooked. The paper presents an example of modeling dependencies between system hierarchy in OMG SysML™ and the CAD assembly in Solid Edge for a mechatronic design task. The dependencies between the two models were represented through a SysML model. The results support the claim that current languages do not provide sufficiently expressive language constructs to model dependencies. Instead, a better solution is to utilize a dedicated Dependency Modeling Language (DML) to capture the dependencies. The dependencies between system hierarchy and CAD assembly are recurring, and can be considered as a reusable dependency pattern. The transformation process (reported in the paper) between Solid Edge and SysML, can be utilized to automate the execution of the dependency pattern between system hierarchy and CAD assembly.

### 7.4 Paper D

*Designing Mechatronic Systems - A Model-Integration Approach*

This paper aims to find out how system-level modeling can support mechatronic design, and how the integration of system-level modeling and domain-specific modeling can be supported during different design phases. A design example of a hospital bed’s propulsion system is presented to show firstly the relationship between conceptual design and system-level modeling, and secondly the need for integration of system level and domain specific design models. The paper presents an integrated modeling and design infrastructure to support abstraction between mechatronic design models, hence supporting co-evolution of design models. The paper concludes that a mechatronic design problem can be better supported through such an integrated design approach.
7.5 Paper E

Integrating Multi-Domain Models for the Design and Development of Mechatronic Systems

In this paper, a model-level integration-framework is described, aiming to identify and solve dependencies across different domains during the design process of a mechatronic system. A two-degree of freedom robot example is presented, to exemplify the iterative process of design optimization in a computer aided design tool (Solid Edge), connected with corresponding dynamic analysis and controller design in Matlab/Simulink. A system model built in OMGs Systems Modeling Language (SysML) is extended, to capture the cross-domain dependencies as parameters in a black-box model. These black-box models can be systematically linked with other domains (CAD and dynamic analysis) through the model integration algorithm. As the modelers perform the design iterations in Matlab/Simulink and Solid Edge, the updates on cross-domain dependencies will be available through SysML model, containing the complete system view.

7.6 Paper F

Designing Mechatronic Systems, A Model-based Perspective, An Attempt to Achieve SysML-Matlab/Simulink Model Integration

Mechatronic design requires careful integration of methods and tools to satisfy the stake-holder objectives. With the advent of systems modeling languages which specify the complete system in one system model, there is an increased urge to link the system modeling tools to domain specific tools such as Matlab/Simulink. In this paper we present an attempt to achieve an integrated design environment by building mapping between SysML and Matlab/Simulink models. Models of an industrial pattern generator are presented to explain the practical influence of this integration approach. The resulting integrated model is more comprehensive for the designer when investigating various design alternatives.
Bibliography


Appended Papers