A BAYESIAN LEARNING APPROACH TO INCONSISTENCY IDENTIFICATION IN MODEL-BASED SYSTEMS ENGINEERING

A Dissertation
Presented to
The Academic Faculty

by

Sebastian J. I. Herzig

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
G.W. Woodruff School of Mechanical Engineering

Georgia Institute of Technology
May 2015

A BAYESIAN LEARNING APPROACH TO INCONSISTENCY IDENTIFICATION IN MODEL-BASED SYSTEMS ENGINEERING

To my Parents, Ulrike and Joachim Herzig

ACKNOWLEDGEMENTS

A dissertation marks the end of a long academic journey. It would not have been possible to complete this journey without the support, guidance and understanding from a great many people.

Firstly, I would like to thank my adviser, Dr. Chris Paredis, for taking me under his wing four years ago, and for his guidance and inspiration over the course of my studies. I would also like to thank my reading committee members – Dr. Leon McGinnis, Dr. Jonathan Rogers, Dr. Rahul Basole and Dr. Tommer Ender – for their careful review of this dissertation and valuable feedback. Furthermore, I would like to thank Boeing Research & Technology for the generous financial support provided throughout my doctoral research.

Secondly, I would like to thank Dr. Ulrich Walter, Dr. Markus Brandstätter and Andreas Hein. Without their support and guidance, mentoring and insightful discussions, and without them sparking and fueling my interest in Model-Based Systems Engineering, I would not be where I am today. In a similar spirit, I would like to thank Florian Hölzl and Dr. Manfred Broy for inspiring and challenging me to think about graphical modeling not just as a communication tool, but as a mathematically well founded concept.

I would also like to thank the members of my laboratory and the Model-Based Systems Engineering Center for the friendship, discussions and moral support. In particular, I would like to thank Dr. Benjamin Lee, Dr. Axel Reichwein, Dr. Ahsan Qamar, Dr. Roxanne Moore, Dr. Kevin Davies, Dr. Aleksandr Kerzhner, William Bailey, William Binder, Jackie Branscomb, Brian Taylor, Douglas Broadwell, Alisha Kasam, Dr. George Thiers and Tim Sprock.

In addition, I would like to thank the various researchers that I have had the pleasure of working and engaging in research discussions with at a number of conferences, workshops and site visits. In particular, I would like to thank Stefan Feldmann, Konstantin Kernschmidt, Thomas Wolfenstetter and Daniel Kammerl, as well as their advisers, Dr. Birgit Vogel-Heuser, Dr. Udo Lindemann and Dr. Helmut Krcmar, all from the Technische Universität München. In a similar spirit, I would like to thank Dr. Dániel Varró, Dr. Rick Salay, Dr. Joachim Denil, Dr. Federico Ciccozzi and Benjamin Kruse. I would also like to thank Dr. Hans Vangheluwe for the guidance and many fruitful research discussions.

Finally, I would like to thank my girlfriend, Megaran Morris, for her love, support and understanding over the course of my doctoral work. Her understanding and dedication to helping me through both good and bad times is more than anyone could ever ask for. Thank you for always believing in me.

Last but certainly not least, I would like to thank my parents. Every journey has a start. Being able to pursue my dreams and being supported, guided and understood to the degree to which I have experienced it throughout my life is a true privilege for which I will be forever grateful and thankful. It is for this reason that I dedicate this thesis to my parents, Ulrike and Joachim Herzig.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

2^A	The power set of a set A (i.e., the set of all subsets of A).
ϵ	Empty word.
I	(Semantic) Interpretation.
$\mathbb{G}$	Directed acyclic graph (DAG).
$\mathcal{D}$	Semantic domain.
$\mathcal{F}$	Symbol denoting a σ-algebra.
$\mathcal{L}$	Formal language.
$\mathcal{T}$	Theory.
MBSE	Model-Based Systems Engineering.
Ω	Sample space of a probabilistic experiment.
ω_i	A single outcome of a probabilistic experiment.
$\otimes$	Composition (of expressions, of languages).
P	Probability measure.
ψ	A sentence (well-formed expression) of a formal system.
Σ	Alphabet: a (typically finite) set of symbols.
$\mathbf{G}$	Directed, labeled multi-graph.
$\mathbf{P}$	Complex graph pattern.
$\mathbf{P}_S$	Simple graph pattern.
TGG	Triple graph grammar.
v	(Semantic) Valuation function (semantic mapping).
w	Word: a concatenation of symbols from an alphabet.

SUMMARY

Designing and developing complex engineering systems is a collaborative effort. In Model-Based Systems Engineering, this collaboration is supported through the use of formal, computer-interpretable models, allowing stakeholders to address their particular concerns of interest using well-defined modeling languages. However, because concerns cannot be separated completely, implicit relationships and dependencies among the various models describing a system are unavoidable. Given that models are typically co-evolved and only weakly integrated, inconsistencies in the agglomeration of the information and knowledge encoded in the various models are frequently observed. The challenge is to identify such inconsistencies in an automated fashion.

In this research, a probabilistic approach to abductive reasoning about the existence of specific types of inconsistencies and, in the process, semantic overlaps (relationships and dependencies) in sets of heterogeneous models is presented. The basis for the approach is Bayesian probability theory. A prior belief about the manifestation of a particular type of inconsistency within a specific context is updated with evidence, which is collected by extracting specific features from the models by means of pattern matching. Pattern matching across heterogeneous models is enabled by translating the information and knowledge encoded in models to a common, graph-based representational formalism. Results of the inference procedure are then utilized to improve future predictions by means of automated learning. The primary focus of the investigation is the development of a mathematically sound framework as a basis for a formal computational method. The effectiveness and efficiency of the approach is evaluated through a theoretical complexity analysis of the underlying algorithms, and through application to a case study. A prototypical, semantic web inspired implementation of supporting software tools was developed as a basis for performing the necessary accompanying measurements. As a case study, randomly generated sets of disparate, heterogeneous models of railway systems are considered. These generated sets of models are algorithmically injected with inconsistencies, and with features representing the result of human error and incompleteness. Numerous experiments are conducted for the purpose of characterizing and evaluating the proposed approach. Insights gained from these experiments, as well as the results from a comparison to a state-of-the-art deterministic reasoning approach have demonstrated that the proposed inexact reasoning method is a significant improvement over the status quo of inconsistency identification in Model-Based Systems Engineering.

CHAPTER I

INTRODUCTION

This dissertation focuses on the topic of identifying inconsistencies in models of engineered systems. Modern technical systems such as aircraft or spacecraft systems are typically developed collaboratively by a great number of people, each using their particular set of interests and skills to address specific concerns by focusing on particular aspects of a system under consideration. This is necessary due to the often overwhelming complexity of technical systems, which makes it impossible for a single human being to understand every aspect in detail.

Allocating concerns is a necessary means of managing the complexity of a larger overall problem by decomposition. However, this decomposition does not completely decouple the various sub-problems assigned to different stakeholders. This necessitates that proper communication paths are put in place and interfaces among the different stakeholders are managed. Not managing these interfaces appropriately can lead to bad decisions being made due to the use of outdated or even misinterpreted information. This can lead to the agglomeration of information and knowledge captured about a particular system being *inconsistent* (i.e., not in agreement and in conflict). Such inconsistencies can result in costly rework, the termination of a project, or, in the worst case, loss of life or mission failure. Therefore, a crucial part of managing interfaces among stakeholders is the early detection of inconsistencies.

In this dissertation, a novel approach to identifying inconsistencies based on Bayesian probability theory is introduced and evaluated. The need for a method for identifying inconsistencies and the reasons for choosing a probabilistic approach are motivated further in the remainder of this chapter. Specifically, in section 1.3, the objectives of the research, investigated research questions and associated hypotheses are introduced. Section 1.4 briefly discusses the strategy used to evaluate the approach. The chapter ends with section 1.5, where an outline of the dissertation is presented.

1.1 Context & Motivation

On December 11, 1998, the National Aeronautics and Space Administration (NASA) launched the USD 200 million *Mars Climate Orbiter* (MCO) mission as part of the Mars Surveyor '98 program. Its mission was to study the Martian weather, climate, and water and carbon dioxide budget. Furthermore, it was to act as a relay satellite for a lander, the *Mars Polar Lander* (MPL), whose mission was to investigate the composition of the soil near the South Polar ice cap on Mars [205]. The orbiter was to enter an elliptical orbit around Mars. Ground control was able to track the spacecraft visually up to the point when it vanished behind the planet. It was expected that the orbiter would reappear shortly after the orbit insertion maneuver. Unfortunately, the spacecraft never reappeared from behind the planet.

It was only several months after the incident that the reason for the failure of the mission was discovered. A peer review led by NASA's *Jet Propulsion Laboratory* (JPL) revealed that a supplier had made different assumptions in regards to what units were to be used for some of the calculations. JPL had used the International System of Units (SI), while the supplier of the ground station had assumed the use of the British Gravitational System (BGI). However, since the supplier assumed that BGI was the system of units used throughout the project, no unit conversions were ever performed. The result was an altitude error of about 150km: behind the planet, the MCO reached an altitude of only 57km as opposed to the intended 226km. The gravitational pull of Mars was too strong for the MCO to escape, and the vehicle burned up in the atmosphere.

Why do seemingly obvious errors such as these remain undiscovered, particularly when measures are put in place to prevent such errors from happening in the first place? NASA's official report states that "[...] contributing causes include inadequate consideration of the entire mission [...] as a total system, inconsistent communications [...] and lack of complete end-to-end verification of navigation software and related computer models". One reason listed as a primary source of the failure was that "some communications channels among project engineering groups were too informal" [151, 205].

The case of the Mars Climate Orbiter is only one of many documented cases, where inconsistent information used in designing and developing a system has lead to costly or catastrophic outcomes. For instance, when Airbus assembled the first Airbus A380 in Toulouse, a pre-assembled wiring harness produced in the Hamburg, Germany plant failed to fit into the airframe [193]. Other examples include the Panama radiation overdose incident [235] and the failure of the Mariner 1 mission [139]. Clearly, there is a need for a mechanism detecting inconsistencies in an automated fashion.

1.2 Inconsistency Identification in Model-Based Systems Engineering

To avoid problems such as those encountered in the case of the Mars Climate Orbiter, *systems engineering* practices such as reviews, tests and, more generally, verification & validation activities are typically employed in the design, development and management of complex systems over their life cycle. Yet, because of the overwhelming complexity of some systems, and the lack of formality and rigor of current systems engineering practices [72], errors, costly rework, and mission failures are still commonplace. However, recent developments in software engineering, and the transition of some of these methods to the domain of systems engineering have inspired the vision of *Model-Based Systems Engineering* (MBSE), which has opened a path towards computer-aided systems engineering practices and has, hence, provided a basis for automated identification of inconsistencies.

1.2.1 Model-Based Systems Engineering

Systems engineering is a multi-disciplinary approach to developing balanced system solutions in response to diverse stakeholder needs [120, 72]. It includes the application of both management and technical processes to achieve this balance and mitigate risks that can impact the success of the project. The management process is applied to ensure that development cost, schedule, and technical performance objectives are met. Typical management activities include planning the technical effort, monitoring technical performance, managing risk and controlling the system technical baseline. The technical processes are applied to specify, design and verify the system to be built [72]. Systems Engineering, as we know it today, began to evolve as a branch of engineering during the late 1950's [120].

While considered mature from a methodological perspective, the tools and methods used in the typically employed document-centric systems engineering approach are informal and ad hoc in nature. For instance, text documents, spreadsheets, informal drawings and presentation slides are commonly used to communicate with other stakeholders, document system architectures, record traceability information and specify subsystem interfaces [72]. Such informal documents are not only difficult to maintain, they are also vague and ambiguous. This is due to a lack of a well-defined underlying formalism (e.g., notation used and meaning of symbols). The lack of a formalism leaves room for interpretation by the stakeholder exposed to the document [24]. For instance, if the notation used in a diagram on a presentation slide is not explicitly defined and understood by the audience, different people may interpret the diagram differently, or even not understand it at all.

Oftentimes misinterpretations of documents are only discovered at decision points or during infrequently held *review* activities. Reviews are activities held mainly for the purpose of verifying & validating the integrity, coherence and consistency of independently evolved parts of a system. Also, during such activities the use of outdated documents may first become apparent. This can lead to costly rework due to bad decisions having previously been made – possibly on the basis of inconsistent information and knowledge. The advent of computational engineering tools and network infrastructures has given rise to the possibility of developing formal, computer-aided methods for this purpose. *Model-Based Systems Engineering* (MBSE) is a recent paradigm shift in systems engineering, where a key principle is the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases [119].

Models are represented in many forms, including physical prototypes, graphics, mathematical equations and logical statements. Within the context of this dissertation, we restrict ourselves to *formal, computer-interpretable models* (henceforth referred to as *models* and *formal models*). Therefore, we do not consider any models that take a physical form. By using the word *formal*, the existence of a relation to some well-defined underlying formalism – such as mathematical logic – is implied. This also implies a well-defined symbolic representation (*syntax*) and basis for interpreting the meaning (the *semantics*) of well-formed syntactic constructs[1] [92]. Kepler's laws of planetary motion (see, e.g., [228]) are examples of models expressed in a mathematical formalism. In systems engineering, functional flow diagrams and schematic block diagrams are examples of commonly used types of models for representing abstract processes or structures. The use of only formal models throughout the life cycle has a significant advantage over the use of informal documents: models can be interpreted by a computer, opening up the possibility to reason about properties of the system under consideration in an automated fashion [72].

[1]For a more elaborate definition of *formal models*, see section 2.1.

1.2.2 Inconsistency Identification & Related Challenges

Two views on the problem are taken in related research efforts: ensuring the absence of inconsistencies by construction, and the identification of specific instances of inconsistencies. In this dissertation, the latter view is taken. The primary reason for taking this view is that previous work has shown that consistency cannot be guaranteed [102].

1.2.2.1 What is an Inconsistency?

Before discussing what *inconsistency identification* in MBSE encompasses, an (informal) definition of *inconsistency* is given. Numerous definitions for, and interpretations of, the term *inconsistency* exist. Inconsistency is typically understood to be a (behavioral) quality, and as the state of being *inconsistent*. This is evident from the definition given by the Merriam-Webster dictionary [118]:

- Not always acting or behaving in the same way
- Not continuing to happen or develop in the same way
- Having parts that disagree with each other
- Not in agreement with something

Merriam-Webster further defines *inconsistent* as the quality of a statement being *"not compatible with another fact or claim"*, an argument *"containing incompatible elements"*, being *"incoherent or illogical in thought or actions"*, and, when referring to a set of mathematical equations or inequalities, being *"not satisfiable by the same set of values for the unknowns"*. In this context, *incompatibility* is defined as *"not being able to exist together without trouble or conflict"* and *"not able to be used together"* as in two propositions *"not both [being] true"*. *Incoherence* is defined as *"not logical or well-organized"*. Within the context of mathematical logic, an inconsistency is also frequently defined as a *logical contradiction* [115].

In the related literature, similar definitions for the term *inconsistent* can be found. For instance, Nuseibeh *et al.* define an inconsistency as *"any situation in which a set of descriptions does not obey some relationship that should hold between them [...] expressed as a consistency rule against which the descriptions can be checked"* [158]. Spanoudakis and Zisman define inconsistencies as *"a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable"* [200].

It is interesting to note that all of these definitions for *inconsistency* share a common property: that a state of *conflict*, marked by the presence of a *contradiction*, *illogical* statement or *disharmony* exists[2]. Indeed, given sufficient information, all of these definitions imply that the state of *inconsistency* is marked by the presence of *contradicting* and *overlapping statements*. Interesting is also that one can argue that the quality of being *inconsistent* can be viewed as an overarching term for numerous other, perhaps more specific types or states of inconsistency of some system such as *incompatibility*, *incoherence* and *unsatisfiability*. From a *value-based perspective*, it can be argue that an inconsistency is marked by a decrease in value (e.g., of the artifact).

1.2.2.2 Inconsistency Identification

Inconsistency identification is the process of *detecting*, *locating* and *classifying* inconsistencies [158]. It is a *"vehicle for integrating views [models]"* [159]. Therefore, inconsistency identification can be viewed as an analysis of (in the case of Model-Based Systems Engineering) formal models. This requires, similar to any other analysis task, knowledge on how to perform the analysis – i.e., knowledge on how to derive an inconsistency from the given model-based description of a system. As will be explained

[2]A more formal definition of the term *inconsistency* as interpreted within the context of this dissertation will be given in section 4.2.2 after having introduced the necessary terminology and background in section 2.

in detail in chapter 5, models can be viewed as encoding information and knowledge (as a set of propositional statements) about the system under consideration. Hence, the knowledge required to identify inconsistencies – the *inconsistency identification knowledge* – must be phrased in such a fashion that inconsistencies can be *derived* from the statements encoded in the models to be analyzed. That is, given a set of models and knowledge about how to identify inconsistencies in these models, one must be able to *formulate an argument* – that is, *reason* – about why an inconsistency is present (or absent) (*detection*), where (i.e., what part of the model(s)) and in what form the particular inconsistency manifests (*locating*), and what class (according to some taxonomy) the particular inconsistency belongs to (*classification*).

1.2.2.3 Challenges in Identifying Inconsistencies

Identifying inconsistencies in MBSE is challenging for a variety of reasons. Firstly, stakeholders create a variety of models and use a number of different modeling languages (often modeling languages specific to their respective domains), which are based on different formalisms. The nature of these models may also vary: some models may be *descriptive* – e.g., for the purpose of *specification* – while others may be *analytical* – that is, are used for the purpose of *analysis*. From a computational standpoint this *heterogeneity* of models presents a challenge, since it hardens the problem of symbolic processing across different models and interpretation of the models. In addition, the sheer complexity stemming from having to interpret and work with a very large amount of data collected during the life cycle of a system has to be taken into account. Analyzing large numbers of models for inconsistencies is challenging, since it requires some level of automation which, in turn, is expected to incur a large computational cost. From a methodological perspective, automating some (or all) of the analysis processes related to identifying inconsistencies is a challenge by itself, since the underlying models are likely to be incomplete, inconsistent and incoherent, possibly leading to erroneous identifications.

Other challenges arise from the fact that, during the design and development of complex systems, multiple versions of models, may exist at the same moment in time. Similarly, the consideration of multiple system variants leads to additional challenges. In both cases, the specific version(s) of each model to be collected and checked for inconsistencies must be identified. In addition to different versions of models, a number of system *variants* may be under consideration at the same time. This poses an additional challenge since each variant by itself should be free of inconsistencies, but multiple variants may be described with one model.

Additional challenges stem from a need to manage the inconsistency identification knowledge: since one can expect the knowledge required to identify all *types* of inconsistencies (that are deemed valuable to identify) to be very large, issues related to ensuring internal consistency and maintenance of the inconsistency identification knowledge become important to consider as well.

1.2.3 Desired Characteristics of an Approach to Inconsistency Identification

Based on the challenges identified in the previous section, a number of desired characteristics of an effective approach to inconsistency identification are presented in the following. These desired characteristics are considered independent of any specific solution approach to inconsistency identification within the context of MBSE.

1.2.3.1 Automated Identification across Heterogeneous Models throughout Life Cycle

In current practice, limited computational support for identifying inconsistencies exists. Modeling tools offer some support for checking the syntactical well-formedness of models (where non-well-formedness is considered an inconsistency). However, inconsistencies spanning multiple models, which, as practice shows, typically have the the greatest consequences associated with them [205, 193], are, to a large extent, only detected by human inspection during activities such as *formal reviews* [150]. Such activities are vital for any verification and validation process, but are also very costly to implement. In addition, because these reviews are typically done very infrequently, inconsistencies are often detected very late in the development process, and at a stage at which (potentially bad) decisions based on this inconsistent information and knowledge may have already been made. Therefore, an objective, and desired characteristic for developing a novel approach to inconsistency identification should be the introduction of a higher degree of automation compared to the status quo.

Automatically checking for inconsistencies across heterogeneous models requires an approach that enables symbolic processing and manipulation of the information and knowledge encoded in the various heterogeneous models describing a system. This includes the need for a mechanism that enables the retrieval of those parts of a set of models that represent manifestations of inconsistencies, and the ability to define semantic relations between models (i.e., model overlap). Ideally, the *identification* of such overlap should also be fully automated.

In addition, in order for an approach to identifying inconsistencies to be effective over the full life cycle of a system, the approach should be capable of handling incomplete models – that is, incomplete descriptions of a system – which are, to some extent, likely to be ambiguous, and a number of properties of the system uncertain.

1.2.3.2 Provision of Rationale & Traceability

An approach to automatically identifying inconsistencies should also provide insight into the *cause* of an inconsistency. That is, it should be capable of providing the rationale that lead to the conclusion that an inconsistency is present. This is necessary for a variety of reasons: firstly, it allows for the *accuracy* of an inconsistency identification mechanism and the identification knowledge used to be evaluated. Secondly, the information can be utilized to implement strategies aimed at avoiding future occurrences of the same, or similar inconsistencies.

Depending on the concrete underlying reasoning method, conclusions may not always be logically correct, but may be *best explanations* or *best guesses*, potentially influenced by heuristics. It is not unlikely that some results determined by an automated reasoning mechanism may seem unintuitive to a human at first glance. Providing rationale allows for a better understanding of why the computational mechanism reached the particular conclusion, and aids in deciding whether a refinement of the reasoning knowledge is required. For a comprehensive inconsistency management strategy, rationale may also be useful for acting on an inconsistency (e.g., how the inconsistency should be resolved, or whether it is worth ignoring it for the time being).

Storing the rationale is not only useful for understanding the reasoning behind the conclusion of an automated inconsistency identification mechanism, but also to identify the underlying cause of the inconsistency. Recent research has shown that the cause may not always be trivial to identify, since a particular inconsistency may be the result of another, which represents the *root cause* [88, 180].

1.2.3.3 Flexible Formulation of Inconsistency Criteria & Knowledge Reuse

The definitions of the term *inconsistency* given in section 1.2.2.1 are fairly abstract. This signifies a great variety of possible inconsistencies, each of which manifests in some identifiable form. Therefore, it is expected that it is possible to identify a set of *types* of inconsistencies, instances of which may be contained in (parts of) models. Such types of inconsistencies may be specific to a language, domain or application. A mechanism that specifically seeks out inconsistencies should be capable of differentiating between different kinds of inconsistencies, and be flexible enough to identify a large variety of inconsistencies.

One approach to this is to explicitly define how different types of inconsistencies manifest in models, and seeking out inconsistencies based on this definition. This leads to the specification of *criteria* that, when fulfilled, indicate the presence of the particular type of inconsistency. A part of the definition of such criteria is the context in which an inconsistency manifests.

1.2.3.4 Extensibility & Continuous Improvement of Accuracy

In addition, an effective approach should consider aspects of maintaining and refining inconsistency identification knowledge. As knowledge about a system and its environment grows, it is expected that knowledge about possible inconsistencies that strongly influence the value of a system also grows. In addition, by analyzing the conclusions reached by the inconsistency identification mechanism, the accuracy and relevance of future conclusions can be improved.

Therefore, an effective approach should define methods for not only acquiring, but also refining existing inconsistency identification knowledge so that performance can be improved over time. This should include the ability to learn from previous conclusions reached – e.g., by making use of the reasoning rationale identified previously as important to store.

1.2.4 Current Limitations & Research Gap

Even though a well-known problem, and explored extensively in the related literature, inconsistency management is still an open research challenge [68, 82, 96, 177]. Particularly the findings in [102, 200] suggest that there is value in performing additional research.

In the related literature, a number of approaches to managing inconsistencies have emerged: these can be broadly categorized as approaches that represent models in a logical database to check for contradictory propositions [71], approaches that actively check (negative) constraints [96, 188], and approaches that make use of procedural or model-based rules to check for inconsistencies [142, 221]. Rule- and model-based approaches, which have been used extensively in more recent related work, have proven to be both effective and pragmatic. However, this is only the case in a limited number of scenarios. In particular, this is due to the inability of rules to deal with unexpected input and the resulting high cost associated with maintaining large sets of complex rules. In addition, rules can lead to erroneous conclusions when the information and knowledge being reasoned over is incomplete, vague, ambiguous or incoherent.

Most of the related research (with very few exceptions: see, e.g., [79, 80, 97, 102, 175]) stems from software engineering research. However, most of the proposed methods expect a (at least locally) *complete* description of a system and, in some cases, an unambiguous mapping to a target *reality*. This is due to these methods relying on *logically correct deductions*. Only recently have methods emerged that take into account incomplete, vague and ambiguous models of software systems for verifying and validating models of (software) systems at early life cycle phases (see, e.g., [186, 65]). However, since descriptions of systems with physical properties are inherently incomplete, this issue is more prominent in MBSE [102]. A gap in the current research is the identification of an appropriate and sound method for reasoning over models of technical systems under these conditions and within the context of MBSE. In addition, most software engineering research is concerned only with a single modeling language and formalism, and is therefore not addressing the additional challenges stemming from the omnipresent heterogeneity of models in MBSE.

While a number of approaches to identifying inconsistencies in an automated fashion are presented in the related literature, the automated detection of a semantic overlap remains an open challenge. Several methods have been reported in the related literature (see, e.g., [200] for an overview), but either require manual input, or make very strong assumptions and are error prone.

In related MBSE research, no methods have been reported for reasoning about inconsistencies and semantic overlap that specifically take the heterogeneity and incompleteness (partiality) of models into account. In software engineering and information systems research, alternatives have been investigated that specifically account for the incompleteness of models: e.g., probabilistic methods [186]. These investigations have lead to promising results. However, such methods have not been applied to semantic overlap detection or inconsistency identification within the context of MBSE.

1.3 Research Objectives & Approach

Based on the previous motivations, the limitations of related work, and the identified research gap, the following motivating research question is formulated:

Motivating Research Question. *How, and to what extent, can inconsistencies in a collection of distributed, disparate and heterogeneous models be identified automatically?*

The primary hypothesis of the research presented in this dissertation is that a probabilistic analysis can overcome the challenges and mitigate the limitations of state of the art approaches identified in sections 1.2.2 and 1.2.4. A basis for the formulation of this hypothesis is the inherent incompleteness, abstract nature and heterogeneity of models used to describe complex systems in MBSE: incompleteness and abstraction imply the presence of *unknown* and *uncertain* quantities and qualities, some of which may be derivable (with certainty) given sufficient knowledge. However, other conclusions (and it is hypothesized that this is the case for most) may not be reachable with certainty given just the state of information at a particular moment in time, and are instead merely *possible* conclusions – that is, they are *uncertain.*

Because the motivating research question is too broad to be answered in a single research study, it merely serves as a starting point to identify several more specific and focused research questions. In this dissertation, three research questions are addressed in an effort to build a basis for a focused investigation. To further narrow the focus and define the scope of the investigation, simplifying assumptions are made explicit in section 1.3.5.

Research Question 1. *What are the characteristics of typical inconsistencies in engineering models? What kinds or types of inconsistencies can be identified, and what unsatisfied semantic relationships are these a result of?*

Research question 1 is aimed at identifying qualities by which the completeness of an approach to inconsistency identification can be measured. One part of answering this research question focuses on identifying characteristics of inconsistencies – that is, how inconsistencies manifest in models. Another part is related to answering the question of what kinds and types of inconsistencies exist, and whether this set is finite.

Closely related to a classification of inconsistencies, and important to investigate, is the identification of the primary *cause* of, and important context information for identifying an inconsistency. This leads to the formulation of the second research question:

Research Question 2. *How can semantic overlap and semantic relationships be identified effectively and efficiently, and to what degree can this be automated?*

One important aspect of identifying inconsistencies is the detection of (semantic) *overlap* among models. This is a non-trivial but essential component of any inconsistency identification strategy, largely due to the disparity and heterogeneity of models. The disparity and heterogeneity, and the fact that models are commonly co-evolved in practice, lead to the third research question:

Research Question 3. *What is an effective way of aiding modelers in the process of efficiently detecting inconsistencies in a set of collaboratively developed, heterogeneous* *and distributed formal engineering models? How can we improve upon the status quo of rule-based approaches?*

The third research question primarily focuses on identifying a suitable approach for identifying inconsistencies under conditions that are common to practical design and development scenarios: co-evolution of heterogeneous models and distributed model repositories. However, it also focuses on the aspect of interpreting the results of an inconsistency identification algorithm for the purpose of notifying a modeler of inconsistencies relevant in the current context.

1.3.1 Characterizing & Classifying Inconsistencies

One important step in developing a comprehensive framework for inconsistency identification is characterizing and classifying inconsistencies. This part of the overall research will address research question 1. The characterization will lead to a (formal) definition of inconsistencies, while the classification focuses on identifying different kinds and, abstracted from that, types (classes) of inconsistencies. Classifying and characterizing inconsistencies is also necessary for the purpose of defining the completeness of the approach and, in part, the scope of the kinds and types of inconsistencies to which the proposed research can be claimed to apply.

In section 1.2.2, a definition for inconsistencies is given. Based on this definition, the following hypothesis is formulated about the characteristics of an inconsistency:

Hypothesis 1. *A state of inconsistency is influenced by the presence (or absence) of a number of syntactic and semantic properties that are in conflict. These syntactic and semantic properties manifest as propositions, and a configuration of conflicting propositions can be abstracted by a pattern. Furthermore, these properties can be understood to represent evidence to suggest the presence (or absence) of a particular type of inconsistency. A conflicting set of such properties (i.e., a match to a corresponding pattern) represents a manifestation of a particular type of inconsistency iff* *(if and only if) it entails the inconsistency.*

A second hypothesis, designed to be a response to the second part of research question 1, is formulated as follows:

Hypothesis 2. *It is possible and practical to differentiate between different types of inconsistencies. There exists both a finite, closed set, and an open, infinite set of types of inconsistencies and related types of semantic overlap.*

Both hypotheses 1 and 2 are investigated in chapter 4. The validity of hypothesis 1 is further explored in chapter 5. To support the hypotheses, findings from expert consultation and results from brainstorming are compared to the related literature. The hypothesis is also further supported by the findings and insights gained from a quantitative, case study driven evaluation. The results of this are presented in chapter 8.

1.3.2 Enabling Symbolic Processing across Heterogeneous Models

In MBSE practice, engineers (and other stakeholders) use tools and create models allowing them to address very specific concerns about specific aspects of a system under consideration. Typically, these models are also specific to a particular domain. While this allows stakeholders to work efficiently and in familiar environments, it also brings about a challenge for consolidating and checking for inconsistencies in the agglomeration of all models. Largely, this is due to the heterogeneity and disparity of the models. Such results in a variety of models, whose underlying concepts and formalism are incompatible, even if two models express similar or equivalent concepts, but in a (syntactically) different manner, or using a different organization of knowledge and structuring of information. This is one reason why model integration infrastructures are not commonplace [2].

Tool chains (or *point-to-point integrations*) are a common practice in MBSE implementations for integrating models, and ensuring the absence of select inconsistencies.

These enable individual models to be integrated with one another within a specific context. However, such infrastructures are very costly to maintain, ad hoc, and fragile [24]. An alternative is offered by the concept of *model transformations*. However, the implementation of such approaches requires making strong assumptions about the organization of knowledge in the model, and necessitates the encoding of a large amount of knowledge in the transformation. Inconsistency identification requires that models can be accessed at any level, and that any part of a model can be retrieved that is deemed relevant to a certain task. Specifically, this is necessary for identifying inconsistencies that involve a variety of models, and for a mechanism for creating relationships among models. In other words, a very flexible mechanism for data retrieval, manipulation and integration is required. Identifying such a method is part of the answer to research questions 2 and 3. The hypothesis is that representing models in a *common representation formalism* that allows for symbolic processing across model boundaries is valuable:

Hypothesis 3. *A prerequisite to an effective method for identifying inconsistencies in heterogeneous models is the transformation of the models to a common representational formalism, thereby allowing symbolic processing across the models regardless of their nature, underlying formalisms, and organization of the encoded knowledge and information.*

Evidence in support of hypothesis 3 is primarily gathered in chapter 5 and, by application to an example, as part of the quantitative evaluation of the approach in section 8.2.

1.3.3 Identifying Inconsistencies Under Uncertainty

One of the important insights from section 1.2.2.3 is that models of systems (at least within the context of MBSE) are inherently incomplete. That is, they are *always* abstractions of reality and incorporate assumptions. In addition, not all information and knowledge is captured explicitly: some may be implicit or tacit. This is particularly the case for decision rationale. Models can also become ambiguous as a result.

For an entity that is not aware of the tacit and implicit knowledge, as well as the assumptions that went into creating a model, arguing about the presence of inconsistencies is difficult. Generally speaking, two options can be considered: either a "yes" / "no" answer is provided that is based on sufficient conditions, or it can be attempted to analyze parts of the given models more closely and derive how probable a particular answer is given the information and knowledge available for reasoning. This leads to derivations and answers to questions such as "is X inconsistent?" being treated as statements with uncertain truth values. A sound basis for such an approach is Bayesian probability theory – and, in particular, the concept of Bayesian updating – where the explicitly available information and knowledge can be used as evidence to support or oppose the probability of a given premise.

Since models evolve over time, vagueness and ambiguity should reduce over time as well, as more information and knowledge becomes available. This leads to beliefs about inconsistencies being updated over time. A concrete method implementing these concepts acts as part of the answer to research question 3. Based on these insights, the following hypothesis about what method an effective approach to inconsistency identification should be based on can be formulated:

Hypothesis 4. *An effective method for identifying inconsistencies throughout the life cycle that is capable of drawing conclusions from an incomplete, but continuously refined description of a system should be based on Bayesian updating.*

The technical feasibility and viability of using a Bayesian approach to reach conclusions from model-based data is investigated in chapter 6. The specific application to inconsistency identification is discussed in chapter 7. Further evidence in support of the hypothesis is collected in chapter 8, where the approach is applied to a case study and compared to a state-of-the-art deterministic approach.

1.3.4 Refining Inconsistency Identification Knowledge over the Life Cycle using Machine Learning

By nature of the approach and the assumptions made, the identification of *probable* inconsistencies suggests reasoning based on incomplete knowledge, and the reaching of conclusions that are not necessarily logically correct. However, as knowledge about a system grows, one may ultimately have a better idea and more accurate depiction about what particular evidence strongly suggests the presence (or absence) of a particular type of inconsistency. Therefore, it is prudent to assume that it is necessary to evolve and refine the knowledge used in identifying inconsistencies over time. In other words, as one learns more about a particular system and its environment, one may also learn more about what kind of inconsistencies can occur in the given context, how they manifest, and what their impact is.

Numerous methods can be envisioned for refining reasoning knowledge over time. Given the previous hypotheses, the application of one particular *(semi-)automated* approach from machine learning – specifically, Bayesian learning – is investigated within the scope of this dissertation:

Hypothesis 5. *An effective approach to inconsistency identification should consider the aspect of learning from experience. Granting hypothesis 4, methods for encoding, integrating and processing relevant past experience and expert knowledge for the purpose of refining inconsistency identification knowledge should make use of (Bayesian) machine learning.*

Hypothesis 4 is primarily evaluated through interpretation of the results gathered from applying the approach to a case study in chapter 8. There, the effect of learning (i.e., giving feedback to the probabilistic reasoning mechanism) is analyzed in detail.

1.3.5 Simplifying Assumptions

As discussed in the previous sections, a very large number of considerations must be made when developing an inconsistency identification approach that accounts for all facets of MBSE. To reduce the investigation to a manageable scope, a number of simplifying assumptions are made throughout the thesis. The main, overarching assumptions are as follows:

- Only those inconsistencies that can manifest as parts of one or more models are considered
- Conclusions are drawn (primarily) from static information and knowledge (that is: only the static content of a model is considered, and not the results of executing, e.g., an analysis)
- Versioning is not considered, and only the latest version of the models that, together, are intended to form a coherent set of models describing a system are considered
- Variant management is not considered

Given these assumptions, no claim about the completeness of the approach developed and investigated in this dissertation is made. However, the results are deemed a significant leap towards a more generally applicable framework.

1.4 Evaluation Strategy

The probabilistic approach to inconsistency identification presented in this dissertation is evaluated both quantitatively and qualitatively. For purposes of quantitative evaluation, proof-of-concept tool support is developed.

In chapter 7 and chapter 8 (specifically section 8.3.1), the developed approach is evaluated qualitatively. This is done through a theoretical complexity analysis of the underlying algorithms, as well as a scalability analysis of the approach in practice, through interpretation of empirical performance measurements. Chapter 7 also evaluates aspects of maintenance and re-usability of the inconsistency identification knowledge.

The developed approach is evaluated quantitatively in chapter 8, where the developed concepts are applied to a case study. A basis for reasoning is provided by the automated generation of sets of heterogeneous models, and by injecting these with random manifestations of inconsistencies, incompletenesses, and features intending to represent the result of human error. Different properties and characteristics of the approach are then analyzed by performing a series of measurements to determine, e.g., the number of identified inconsistencies, and the accuracy and precision of the method. Impacts of choosing different scopes of reasoning knowledge, (automated) refinement of the knowledge and changes to the boundary conditions are also investigated.

As part of the evaluation, the approach is also compared to a status-quo (deterministic) reasoning approach by comparison of the incurred costs of each (which, as is explained in detail in the chapter, is an indicator and basis for the comparison of the *value* (i.e., utility) of the approaches). This gives insight into the overall value of the approach, and the conditions under which (significant) improvements over the application of status-quo methods can be achieved.

It should be noted that a full *validation* of the approach is deemed impossible (particularly based on insights gained from previous research [102]). Validation entails the consideration of every possible application scenario and circumstance under which the approach could possibly be utilized. This is impossible to do within the scope of a single dissertation. Instead, the approach is *evaluated* under specific conditions.

1.5 Outline of Dissertation

The remainder of this dissertation is organized into three major parts: *background and related work*, both of which act as a foundation for the *development of a probabilistic reasoning method and its application to inconsistency identification.* Finally, the last part of the dissertation is concerned with the *evaluation of the developed concepts.* More specifically, the remaining chapters of this dissertation are organized as follows:

- Chapter 2 introduces important background on automated reasoning in formal systems, formal modeling languages and Bayesian probability theory.

- Chapter 3 provides on overview of the related literature on inconsistency management, and introduces the state of the art of inconsistency identification and semantic overlap detection in MBSE, software engineering and related disciplines.

- Chapter 4 introduces fundamental aspects of inconsistencies, and presents their characterization and classification, as well as a framework for inconsistency identification in MBSE.

- Chapter 5 presents a conceptual basis for a common representation formalism allowing the capture, retrieval and manipulation of information and knowledge encoded in heterogeneous models.

- Chapter 6 is one of the most important chapters of this dissertation. It introduces a novel approach to inexact, probabilistic reasoning. The approach uses the concepts developed in chapter 5 as a basis for implementing the initial ideas from chapter 4.

- Chapter 7 represents the second most important chapter of this dissertation. In the chapter, the application of the concepts developed in chapter 6 to inconsistency identification and semantic overlap detection are discussed. The chapter presents important implications of using inexact reasoning for the identification of inconsistencies.

- Chapter 8 illustrates the value and power of the developed inconsistency identification method through application to a case study and examination of the performance of the approach.

- Chapter 9 summarizes the main insights gained and contributions made by this dissertation.

- Appendix A lists important material used in the process of evaluating the approach.

Readers familiar with the concepts of *formal languages*, *formal systems automated reasoning* may want to skip the first part of chapter 2. Readers primarily interested in the developed inexact reasoning method should focus on chapters 5 and 6. If the application to inconsistency identification and semantic overlap detection is of interest as well, it is recommended to first read chapter 4 (specifically section 4.3). The application of the reasoning method to inconsistency identification and semantic overlap detection can then be found in chapter 7. Results from applying the method, as well as important insights gained are presented in chapter 8.

CHAPTER II

BACKGROUND

In this chapter, relevant background material is introduced. The chapter starts with the introduction of *formal languages* and *formal modeling languages.* Thereafter, automated reasoning in *formal systems* is introduced, which includes a discussion on the interplay with semantics. To provide a formal and mathematically sound basis for inexact reasoning, Bayesian probability theory and related aspects of *machine learning* are introduced thereafter.

The primary purpose of this chapter is to introduce a number of relevant and important concepts from *formal methods.* Formal methods employ techniques from mathematical logic and discrete mathematics for the development, specification, verification and construction of systems [206, 127]. This allows one to write down the behavior of the system under consideration using a well-defined mathematical formalism and to express properties about the system. While this requires all assumptions to be explicitly addressed, it also allows conclusions to be drawn (i.e., *reasoning* to be done) within the given formal framework. Understanding automated reasoning within the context of formal models requires a brief introduction into some of the basic concepts related to *formal languages* and the constituents of a modeling language. The second part of the chapter then introduces Bayesian reasoning, which is used as a basis for the *inexact* reasoning method for reasoning over formal models that is introduced with this research.

2.1 Formal Models & Modeling Languages

In contrast to documents written in natural language, formal methods are based on *formal languages* and require the explicit and concise notation of all assumptions. An understanding of formal methods and formal languages is essential to understanding automated inconsistency identification.

This section is not intended to give a complete introduction to the vast field of formal methods, but merely introduces the most important concepts which aid in understanding the remainder of this dissertation. For greater depth, a variety of books and articles on formal methods are available (e.g., [31, 44]).

2.1.1 Formal Languages

A formal language $\mathcal{L}$ is a set of *well-formed* syntactic expressions. These well-formed expressions are constructed from concatenations of *symbols* from some *alphabet* Σ. Formal languages are often represented in a compact form using a *grammar*. Grammars can be analytical or generative. *Analytical grammars* are used in deciding whether an arbitrary sequence of symbols is a well-formed expression. *Generative grammars* define a set of production rules that, when applied, generate only well-formed expressions – i.e., elements of the language. Meaning can be given to utterances of a formal language through the definition of language *semantics*. Examples of formal languages include the set of all Java programs, and the set of all words from the dictionary. Note that many of following notations and definitions are adopted from [115].

2.1.1.1 Definitions

In the following, definitions relevant to formal languages are introduced. An *alphabet* Σ is a finite set of symbols. A *word* over Σ is a finite sequence of symbols from Σ. The empty word is typically represented by the symbol ϵ. Words are produced through concatenation of symbols: if u and v are words, then uv is their concatenation. Likewise, if w is a word, then w^n is defined by $w^0 = \epsilon$ and $w^{n+1} = ww^n$. For example, $(ab)^3 = ababab$. The *length* of a word – i.e., the number of symbols – is denoted by $|w|$ [115].

Σ^* is the set of all possible words over Σ, where $*$ is the Kleene-star [115]. A formal language over an alphabet Σ is a subset $\mathcal{L} \subseteq \Sigma^*$. Note that Σ^* is countable if Σ is finite – that is, if one can define a bijective mapping between elements of the set Σ^* and a subset of $\mathbb{N}$ (the natural numbers). Many well-known languages, such as most logics, are *not* countable due to the arbitrary number of symbols representing variables [115].

There are two basic constituents to any formal language: *syntax* and *semantics*. Syntax is the study of producing *sentences* of a language – that is, the study of how symbols are concatenated to produce valid elements of $\mathcal{L}$. Semantics, on the other hand, is the study of *meaning*. That is, semantics focuses on *interpretation* of utterances of a language.

Note that, in the following, the terms *word*, *sentence*, *statement* and *well-formed formula* may be considered equivalent. Also note that alphabets are not restricted to single letters, but can be considered a general set of expressions. For this reason, the term *vocabulary* and *alphabet* may be used synonymously.

2.1.1.2 Abstract & Concrete Syntax

The term *syntax* is used whenever one refers to the *notation* of a language. Syntax focuses purely on notational aspects, and disregards meaning.

Traditionally, syntax is partitioned into *abstract* and *concrete* syntax. Abstract syntax is a machine's internal representation and captures the *essence*. That is, any details that are unnecessary in order for a machine to be able to compute with a syntactical object are removed. Abstract syntax is independent of any particular representation. Concrete syntax, on the other hand, adds additional details to the abstract syntax that are used for representation purposes (typically for human consumption).

In computer languages, abstract syntax is typically stored in a tree structure called the *abstract syntax tree.* Concrete syntax contains additional details that may be implicit to the tree structure or unnecessary: for instance, most programming languages feature parentheses for grouping expressions. A *transformation* to an abstract syntax in the form of a tree structure typically removes these parentheses due to the grouping being inherent in the tree structure.

2.1.1.3 Syntactic Structure & Well-Formedness: Regular Expressions & Grammars

Grammars are compact notations for formal languages. There are two main types of grammars: analytical and generative grammars. Analytical grammars are used in determining or *checking* whether a *given* expression is a valid utterance of a corresponding formal language. Generative grammars, on the other hand, are used in *producing* expressions of a language. Expressions that are valid utterances of a formal language are also referred to as being *well-formed.*

In the following, two mechanisms for generating and analyzing expressions of a (decidable) formal language are introduced briefly: regular expressions and context-free grammars [115, 129]. Regular expressions are compact notations using which *regular languages* can be described. Context-free grammars are grammars capable of producing exactly the set of *context-free languages.* According to the Chomsky hierarchy, all regular languages are also context-free, and both regular languages and context-free languages are subsets of all decidable languages. Therefore, a context-free grammar can be defined for any regular expression.

Regular expressions (REs) are defined inductively:

- For every $a \in \Sigma$, a is a regular expression
- If α and β are regular expressions, then so are $\alpha\beta$, $\alpha|\beta$ and α^* (and β^*)

Note that, as before, $\alpha\beta$ denotes the concatenation of two expressions – in this case, *regular expressions.* $\alpha|\beta$ is used for representing an "or" decision: for the case of generating expressions, either α or β may be used. Analytical grammars allow for either expression to appear in the input word. $*$ denotes the Kleene-star. Regular expressions allow for the use of parentheses to group regular expressions. In grouping, the rule applies that the binding of the Kleene-star is greater than concatenation, whose binding is greater than that of parentheses.

The languages described by regular expressions are defined recursively:

- $\mathcal{L}(a) = \{a\}$
- $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$
- $\mathcal{L}(\alpha|\beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$

A language is regular iff a *discrete finite automaton* (DFA) can be constructed that accepts it [115]. Therefore, any regular language is also decidable. All finite languages are regular [115]. In practice, regular expressions are often used as a means for pattern matching, particularly for textual expressions. For this purpose, automatons are constructed that accept words conforming to exactly the regular expression. This is detailed in [115].

Not all languages are decidable. Of those that are decidable, only some are regular. A larger set of decidable languages are *context-free languages*. Context-free languages can be described using context-free grammars.

A context-free grammar is a tuple $G = (V, \Sigma, P, S)$, where V is a set of non-terminal symbols (also referred to as *variables*), Σ is an alphabet of *terminal symbols* (disjunct with V) and $P \subseteq V \times (V \cup \Sigma)^*$ a finite set of *productions*. $S \in V$ is the *starting symbol*. As a convention, non-terminal symbols are denoted by capital letters from the alphabet – i.e., $A, B, C, ...$ – and terminal symbols are denoted by lower case letters and special characters. Letters from the Greek alphabet are used to denote expressions formed by $(V \cup \Sigma)^*$. Productions are *replacement* rules, which replace a given left-hand side with a right-hand side. Productions are written $A \to \alpha$ rather than $(A, \alpha) \in P$. Furthermore, the notion of the logical "or" is adopted to write $A \to \alpha_1 \mid \alpha_2$ as a shorthand for the two separate productions $A \to \alpha_1$, $A \to \alpha_2$.

An example of a context-free grammar for arithmetic expressions can be constructed in the following way (this example is taken from [115]): Let $V = \{E, T, F\}$, $\Sigma = \{a, +, *, (,)\}$ and $S = E$. Furthermore, let:

$$P = \left\{ \begin{array}{l} E \to T \mid E + T \\ T \to F \mid T * F \\ F \to a \mid (E) \end{array} \right\}$$

A *chain* of productions can then be applied to form expressions such as a or $a * (a + a)$. This can be formalized to the general principle of an induced *deduction relation* in G denoted as $\to_G$: $\alpha \to_G \beta$ – that is, β can be *deduced* from α. This is the case whenever there exists a production $A \to \gamma$ in P, and words α_1, α_2 such that $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$. For instance, relating back to the example of arithmetic, let $\alpha = a + T + a$ and $\beta = a + T * F + a$ (where $\alpha_1 = \alpha_2 = a$). Then β can be deduced from α – that is, $\alpha \to_G \beta$ – because of the production $T \to T * F$.

Chains of deductions – i.e., deductive relations over multiple productions – are denoted using the conventions for expressing the *reflexive transitive hull*: $\alpha \to_G^n \beta$, where $n \geq 0$ is an indicator of the number of production steps. For $n = 0$: $\alpha \to_G^0 \alpha$, and generally for an $n > 0$ one uses the notation $\alpha \to_G^+ \beta$. In spirit with the definition of the Kleene star, one denotes $\alpha \to_G^* \beta$ for $n \geq 0$ productions.

Given a context-free grammar G, the language produced using G can then be defined as $L(G) = \{w \in \Sigma^* \mid S \to_G^* w\}$. Note that, in general, there exists infinitely many grammars for a particular, given language[1].

[1] A simple way of thinking about this is to add productions to a grammar that simply concatenate the empty word.

2.1.1.4 Semantics

Syntax is concerned with purely syntactical aspects of a formal language. Syntactic expressions are used for communicating *information* [91]. Often, the meaning of such information is "obvious" to a user from the syntactic representation alone. This is because syntactic expressions are used that a human is familiar with and can easily interpret (e.g., functions from the standard library in C, or API function names are good examples of this). However, in order for a computer (and other people not familiar with the terms used) to do something useful with the expression, it needs to have the same *semantic interpretation* of the syntactic structure. Semantic interpretation is especially problematic because sometimes two *semantically identical* things can be expressed in two different ways syntactically: e.g., "01/17/1986" and "The third Friday in the first month of the year 1986" refer to the same date, but are syntactically very different. Ideally, in the process of communicating information, both communicating entities must interpret syntactic expressions of a language in exactly the same way. Formally, meaning can be given to utterances of a formal language by defining *formal semantics.*

Broadly, for an arbitrary sequence of symbols, semantics assign the meaning of the expression being *sensical* or *non-sensical* under the given interpretation. The semantic definition for a language $\mathcal{L}$, or simply semantics, consists of a semantic domain of discourse $\mathcal{D}$ and a semantic mapping $v : \mathcal{L} \rightarrow \mathcal{D}$ from the syntax to the semantic domain. Using the example of arithmetic expressions, the meaning of an arithmetic expression α would be a number. Therefore, one possible definition for the semantic domain is $\mathcal{D} = \mathbb{N}$. The semantic mapping (sometimes also referred to as the *valuation function*) can then be defined by $v : \alpha \rightarrow \mathbb{N}$. Such mappings can often be defined in an inductive fashion by providing the meaning of complex expressions in terms of meanings of simpler expressions (e.g., to represent the meaning of the addition of two numbers): for instance, $v(``a+b'') = v(``a'')+v(``b'')$, where the symbol

"+" is mapped to mathematical addition. This is a way of defining *compositional semantics* [165]. For most (if not all) languages, mathematics provides a standard semantic domain as a basis for semantic mapping.

Semantics are not only useful in defining *behavioral* aspects [92]: both behavior and structure need semantics. Behavioral semantics are typically much harder to define. This is because deciding upon a specific semantic domain of discourse requires deciding upon the kinds of things we want our language to express. For example, standard semantic domains used for describing behavioral semantics are trace semantics [110], input/output relations [143], and streams and stream processing functions [25].

A semantic domain specifies the very concepts that exist in a universe of discourse and is a prerequisite for comparing different semantic definitions. Therefore, the explicit and formal definition of a semantic domain is crucial. However, in practice this is often (if not mostly) not the case: for instance, the *Unified Modeling Language* (UML) allows for semantic variation points, but no formal way of defining semantics [91]. In some cases it is also not required: the degree of formality used when defining semantics depends on the intended audience (for instance: users, language developers and tool vendors) and can be, at least partly, a sociological process [92].

2.1.1.5 Meta-Languages

As shown in the previous sections, certain conventions and symbols are used in defining languages. These conventions and symbols are a part of another language. Therefore, a language must exist using which other languages can be described. Tarski first published this observation by proposing the following [214]: *"If the language under discussion (the object language) is $\mathcal{L}$, then the definition should be given in another language known as the meta-language, call it $\mathcal{L}_{\mathcal{M}}$"*. A meta-language can be defined as a language for describing grammars, where a grammar is a sentence in the formal meta-language. At least one well known [115] standardized language for writing context-free grammars is the *Backus-Naur Form* (BNF).

Note that a meta-language is not only required for syntax, but also for semantics – that is, a syntactic representation of a semantic domain and semantic mapping. This is required in order for a computer to process formal semantics. Therefore, to properly define the semantic domain and semantic mapping, a language for describing each in is required, too. This is problematic, since the syntax and semantics of a meta-language must be defined through a meta-meta-language, and so on. This chain of meta-languages terminates if a language is capable of describing itself (a so called *bootstrapping* language). In the literature, a variety of rigorous notations for semantic domains exist. Note that these have varying degrees of formality: expressing semantics in a natural language is an informal, but often sufficient way. Often used are also the language Z [201] or pure mathematics. For languages utilizing graph structures, *graph transformations* are a formal way of defining semantics and semantic mappings [115, 57].

2.1.2 Modeling Languages & Meta-Modeling

Given the definitions from the previous section, the notion of a *modeling language* can now be formalized. Modeling languages are sometimes also referred to as *iconic* or *diagrammatic* languages due to their graphical syntax. Modeling languages can be more intuitive than textual languages, where basic syntactic expressions are put together in linear sequences. However, modeling languages can also be *confusing* if the icons are used in abundance [91] (hence making a compositional mechanism for both syntax and semantics similar to that described for compositional semantics of textual languages desirable).

The definition of a formal modeling language is practically identical to that of a (textual) formal language. Indeed, from a theoretical perspective, there is no principal difference between textual and diagrammatic languages [91]. Similar to the definition of formal languages, a modeling language is a set of utterances, which can be produced through, or verified for well-formedness by a corresponding grammar. The primary difference to other formal languages is that modeling languages typically have a visual syntax (e.g. graphical, geometrical or topological) as opposed to textual [82].

Both the abstract syntax and the concrete syntax of a formal modeling language is typically defined (at least in part) by a so-called *meta-model*. A meta-model is a model itself and, hence, must conform to a meta-modeling language. There are a variety of ways to define meta-models for modeling languages. A commonly used approach is to treat a meta-model as a *type graph*. Elements of the language described by such a meta-model are then *instance graphs*. In order for the particular model to conform to the given language, there must be a morphism – i.e., a *structure preserving mapping* – between an instance graph (the model) and the given type graph (the meta-model). In practice, *Entity Relationship Diagrams* (ERD) and *Class Diagrams* (which add inheritance to ERDs) are commonly used as meta-modeling languages [82]. Due to their limited expressiveness, these are commonly used in combination with a separate, often textual, constraint language (one example of such a language is the *Object Constraint Language* (OCL) [164]).

A second, more general approach defines a meta-model through a set of production (or (model) *transformation*) rules. These can be used two-fold: transformation rules can be used to generatively build a model (similar to generative grammars introduced in section 2.1.1.3) by applying a series of transformations either to an existing, non-empty, or to an empty model. Transformation rules can also be used to analytically check whether a particular given model is a valid utterance of a formal language through the process of *reduction*[2]. Commonly used for this purpose are

[2]In mathematics, reduction is the process of rewriting an expression (in this case, a graph) to a

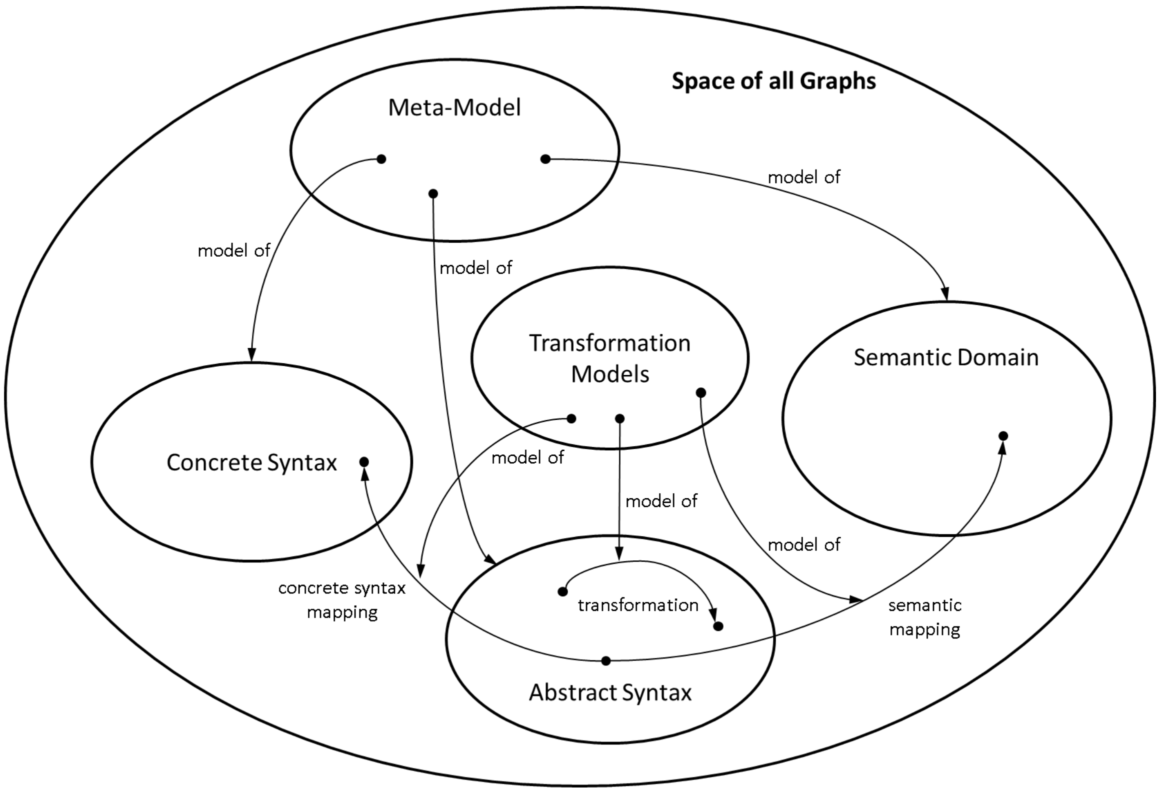


Figure 1: Modeling languages as sets of graphs (adapted from [82]).

graph grammars [57, 132], particularly due to their expressiveness and formality.

Since meta-models are models themselves, they are also defined by a modeling language. The meta-model corresponding to a given meta-model is then referred to as the *meta-meta-model*. The meta-meta-model has the same definition as a meta-model. Therefore, the term *meta* is relative, and one could continue the meta-hierarchy infinitely [82]. However, in practice this is commonly done to a point where a modeling language can be used for the purpose of describing itself – i.e., the language is self-referential and can bootstrap itself (see, e.g., the *Meta-Object Facility* (MOF) [163]).

As mentioned in section 2.1.1.4, the semantic domain and semantic mapping for a given language whose syntax is based on graph structures can be defined using graph transformations (or graph production rules), given that the semantic domain can be expressed using graph structures as well. Since both the syntactic utterances and the

simpler form.

corresponding grammar of a modeling language can be described using type graphs, instance graphs and graph transformations, a natural way for describing the semantic mapping of modeling languages is using graph transformations [91].

Figure 1 summarizes the application of meta-models and transformations for describing both syntax and semantics of modeling languages.

2.2 Automated Reasoning in Formal Systems

In this section, fundamentals of, and methods related to *automated reasoning* (i.e., the process of automating *deductions* (see section 2.1.1.3)) are introduced. Automated reasoning (or, as it is often referred to, *theorem proving* with computer support) concerns the mechanization of *deductive reasoning* within a *formal system.* In practice, a variety of automated theorem provers that are based on numerous proof methods (e.g., direct proofs (similar to applying production rules to find an expression), mathematical induction, or proof by contradiction) have been developed (e.g., SPASS [230], Isabelle [155] and Gandalf [213]).

The following sections briefly summarize important principles of automated proof theory and principles related to automated reasoning. Towards the end, important properties of formal systems are introduced.

2.2.1 A Simple First-Order Language

To illustrate the concepts of automated proofs, and to act as a reference for a simple logical formalism in later chapters, a simple first-order language is introduced in the following. The language is based on the first-order language introduced in [189].

Consider an alphabet Σ which defines variables $(X_1, X_2, ...)$, constants $(a_1, a_2, ...)$, (syntactic) function symbols $(f_1, f_2, ...)$, logical connectives and quantifiers $(\vee, \wedge, \neg, \rightarrow, \forall, \exists)$, and punctuation symbols $((,), ``,'')$ (where the last character is a comma). A *term* t_i can be constructed from elements of this alphabet using the following rules: (1) variables and constants are terms and (2) if f_i is a function symbol and $t_1, t_2, ..., t_n$ are terms, then $f_i(t_1, ..., t_n)$ is a term. Let the set of all terms be constructed by these two rules. Since, in a (first-order) logic, terms are typically interpreted as objects (that is, *things which have properties*), predicate symbols are used as a means of making assertions. An atomic *formula* is defined as $p_i(t_1, ..., t_m)$. These atomic formulas are the simplest expressions in the language. More complex formulas can be constructed using the logical connectives.

In the language, any *well-formed formula*[3] (wff) is defined by the following two rules: (1) any atomic formula is a wff and (2) for wff's A and B, the formulas $\neg A$, $A \vee B$, $A \wedge B$, $A \rightarrow B$, $\forall X_i(A)$ and $\exists X_j(A)$ are well-formed.

2.2.2 Model Theory

Formal languages are entirely syntactic in nature but may be given semantics that give meaning to the elements of the language. For instance, in mathematical logic, the set of possible formulas of a particular logic is a formal language, and an interpretation assigns a meaning to each of the formulas – usually, a truth value. A basis for semantic interpretation, and the relation of semantics to formal languages has already been introduced in section 2.1.1.4. This initial discussion is built on in the following.

As mentioned in section 2.1.1.4, the study of interpretations of formal languages is called *formal semantics*. Semantics of formal (modeling) languages are typically defined in terms of model theory [189, 29]. Of particular importance are the concepts of logical consequence, validity, completeness, and soundness. In model theory, the terms that occur in a formula are interpreted as mathematical structures (e.g., groups, fields or graphs), and fixed compositional interpretation rules determine how the truth value of the formula can be derived from the interpretation of its terms.

Model theory works with three levels of symbols: logical constants, variables and symbols which have no fixed meaning, but are assigned meaning by being applied to a

[3]Note that in most logics, it is customary to refer to utterances of a logic as a *formula*.

particular structure. Examples of the latter are non-logical constants such as relation symbols and function symbols, as well as quantifiers. As introduced in section 2.1.1.4, an interpretation of a wff requires a domain of discourse $\mathcal{D}$ and a mapping relative to $\mathcal{D}$ which assigns semantic meaning to each well-formed constituent of that formula to be defined. For instance, for a logic, `TRUE` or `FALSE` can be assigned to a wff.

Once again referring to the example of a logic, a *model* for a wff is an interpretation I of terms such that the formula becomes `TRUE`. Determining this truth value requires additional information: say a wff A is given. If information is provided that allows one to assign a truth value to A, then A is said to be *interpreted*. A language is interpreted if there exists some systematic way (e.g., using semantic rules) to interpret each wff of a language. Given such a systematic way of interpreting wff's, one can calculate the truth value of each formula (under the given interpretation). In model theory, a formula A is said to be *satisfiable* under interpretation I if there exists *at least one* valuation v for which $v(A)$ evaluates to `TRUE`. A is said to be *valid* if every valuation for every I yields `TRUE`, in which case A is referred to as a *tautology* [189].

Referring back to the example of mathematical addition from section 2.1.1.4, as well as using the first-order language defined in section 2.2.1, consider the term $plus(X_1, X_2)$. Note that *plus* represents a function symbol with two variables X_1 and X_2 as arguments (where, as defined in section 2.2.1, variables are also terms). The valuation of the term requires mathematics as a semantic domain. The semantic mapping is defined in such a way that *plus* maps to the mathematical operator of addition, and the values bound to the variables are interpreted as numbers in the semantic domain. Therefore, for $v(X_1) = three$ and $v(X_2) = four$, where $three$ and $four$ are constants, $v(plus(three, four)) = v(three) + v(four) = 3 + 4 = 7$. Note the compositional definition of semantics as discussed in section 2.1.1.4. Now consider the wff $v(plus(X_1, four)) > six$, where $v(six) = 6$. This wff is *satisfiable* because there exists at least one *model* – i.e., one interpretation – for which the formula is true (e.g., $v(X_1) = \mathit{three}$). However, the formula is not considered *valid* – that is, it is not a tautology. This can be shown through a proof by contradiction: let $v(X_1) = \mathit{one}$ under a given interpretation (where $v(\mathit{one}) = 1$). Then $v(\mathit{plus}(X_1, \mathit{four})) > \mathit{six}$ is no longer satisfied.

Also note that only the basic principles of model theory required for understanding the argumentation in the remainder of this dissertation are discussed. For more a more elaborate introduction to model theory, the interested reader is invited to consult the related literature (e.g., [29]).

2.2.3 Formal Systems, Theories & Reasoning

A formal system (typically also referred to as a *calculus*) consists of a formal language and a deductive system. A deductive system is composed of axioms (i.e., statements of a language which are taken as factual) and a set of *inference* rules. These inference rules are used in deriving further statements (called *theorems*). The set of axioms and inference rules is called a *deductive system*, a set of axioms with all derivable theorems from it is a *theory* $\mathcal{T}$. All elements of a theory are well-formed expressions of a language. A *proof* is a series of purely syntactic transformations according to the inference rules. Therefore, the concept of a formal theorem is fundamentally syntactic. $\vdash \psi$ (where ψ is a formula) means that ψ is a theorem in the given formal system (that is, ψ is provable from the axioms). Sometimes the axioms used as a starting point are made explicit by stating the initial *assumptions* $\mathcal{A}_i$. In this case, and in case additional assumptions are made, one writes $\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n \vdash \psi$ instead.

As mentioned in section 2.1.1, statements of a language may be broadly classified into nonsense and well-formed expressions. Well-formed expressions are typically divided into theorems and non-theorems. However, most formal systems simply define all well-formed formulas as theorems [113]. This subdivision of concatenations of symbols from an alphabet into nonsensical expressions, well-formed expressions and

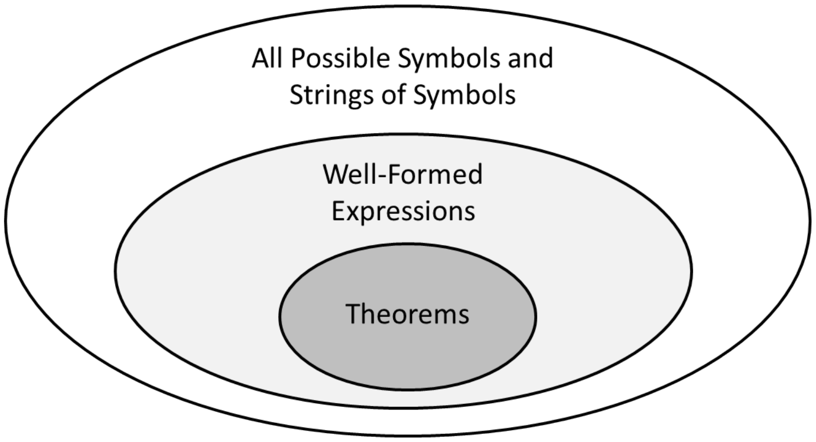


Figure 2: Types of syntactic entities constructable from a given alphabet.

theorems is depicted in figure 2.

The concept of a formal theorem is fundamentally syntactic. This is in contrast to the notion of a sentence that is *true* (a sentence with a truth value is also called a *proposition*), which introduces semantics: depending on the presumptions of the derivation rules, different deductive systems can yield different interpretations. This shows the interplay between proof theory and model theory: let I be a set of interpretations for a calculus and ψ be a sentence (i.e., well-formed expression) of the calculus. As mentioned in section 2.2.2, ψ is satisfiable (under I) if and only if at least one interpretation of I valuates ψ to `true`. ψ is (universally) valid, written $\models \psi$, if and only if *every* interpretation in I valuates ψ to `true` [127]. Model-based deduction techniques use algorithms which try to systematically test all valuations of a formula (propositional satisfiability test, model checking, etc.). This is only possible if the domain is finite (or has finite model properties) [27].

2.2.3.1 Forward and Backward Chaining

A classic example to illustrate the process of a proof, and its purely syntactic nature, is that of proving that *Socrates is mortal* [81]. Let *mortal* be a property of the object *Socrates*. Then, using the same formalism introduced in section 2.2.1, one can express the fact that *Socrates is mortal* in the following way:

$$\text{mortal(Socrates)}$$

Let the above expression be the term t. A proof is now constructed to determine whether the above statement is a logical conclusion that can be reached using the axioms and inference rules of the formal system. To do so, let the formal system under consideration be composed of the following axiom:

$$\text{man(Socrates)}$$

Furthermore, let the following inference rule be a part of the formal system. The inference rule is written in the form of a logical implication, which is, essentially[4], a production rule which states that if the left hand side is true, then the right hand side must also be true.

$$\text{man}(X) \longrightarrow \text{mortal}(X)$$

The fact that *Socrates is mortal* can be proven using a variety of methods. One method consists of producing all possible statements by starting from the axioms and applying inference rules until the statement to be proven has been produced. In that manner, one starts off with the axiom man(Socrates). Then the inference rule is applied, where X is a variable which, when assigned the value Socrates leads to the implication man(Socrates) $\longrightarrow$ mortal(Socrates). Since the left hand side of the implication is known to be true (the known axiom), the statement on the right hand side can be *inferred* to be true also. This inferred statement corresponds to the statement to be proven. Hence, it has been proven that mortal(Socrates) – i.e., that *Socrates is mortal.*

[4]Production rules and logical implications are not quite the same concept, but they are highly related: in a production system such as that used in section 2.1.1.3, production rules were used to construct all utterances of a language. This is a generative concept. Here, such rules are employed for the purpose of reaching *logical conclusions given a set of premises* through *inference*, and proving that a given statement is a theorem. Although similar in application, the connotation is slightly different in that an expression is never explicitly produced, but its *producability* verified.

Starting from the axioms and applying inference rules until a target theorem has been derived is also referred to as *forward inference* or, if multiple rules of inference are applied sequentially, *forward chaining*. It is one of the two main methods used in practice for performing inference – or *reasoning*. Logically, it is a repeated application of the *Modus Ponens*. The opposite of forward chaining is *backward chaining*. In backward chaining, the set of inference rules is searched until a rule is found whose *consequent* (i.e., right hand side) matches the statement to be proven. The *antecedent* (i.e., left hand side) is then compared to the set of axioms. If no match is found, the process is repeated by attempting to prove the antecedent of the rule. Ultimately, a successful proof results in the sequence of backward inductions terminating with an axiom.

2.2.3.2 Deductive, Inductive and Abductive Inference & Monotonicity in Reasoning

Deductive, inductive and abductive inference are the three major types of inference [46]. In deductive inference the inferred statements are necessarily true if the premises from which it is inferred are also true. That is, deductive inferences are based on relations between premises that are logically valid. Therefore, the truth of premises guarantees the truth of conclusion. This is similar to the example used in the previous section, where the statement *Socrates is mortal* was inferred from the premises that *Socrates is a man* and *all men are mortal*. Reasoning based on deductive inference is sometimes referred to as *exact reasoning*.

Inductive and abductive inference differ from deductive inference in that the conclusions drawn are not guaranteed to be correct – that is, the conclusions are *uncertain*. For instance, consider the premises *Socrates is a philosopher* and *Most philosophers do not believe in free will*. Using inductive reasoning, one can now conclude that *Socrates does not believe in free will* based on the given premises. However, the truth of the conclusion is not guaranteed – in fact, in this case, the conclusion drawn is wrong, since Socrates was an advocate of the concept of free will (see, e.g., his dialogue “Phaedo” [224]). Therefore, the concept is often seen as being *statistical* in nature. Abduction is a similar concept (and some conceive induction as a special case of abduction [93]), but is explanatory in nature. In [46] the example is given that one may have observed many gray elephants, but no non-gray ones. From this one infers that all elephants are gray, because it provides the *best explanation* for the observations made. This explanatory link is absent in inductive inference. Explanatory reasoning also allows for *diagnostic* reasoning by exploiting the causal links. Reasoning based on inductive or abductive inference is sometimes referred to as *inexact reasoning*.

Deductive inferences are monotonic: that is, once it has been established that a derived premise is true, no other inferences can prove otherwise (unless the deductive system is *inconsistent*[5]). Inductive and abductive inference, on the other hand, have the interesting property that they are not monotonic. That is, there are conclusions that cannot be drawn from a formal system as a whole, but they can be drawn when considering only a subset of the premises.

Theorem provers typically use deductive inference due to the intended use of producing a proof that is guaranteed to be correct under fixed assumptions. However, much research has also been conducted in automated reasoning using inductive and abductive inference. Typically, due to the inherent uncertain nature of the conclusions, inductive and abductive inference is typically based on methods such as *Bayesian reasoning* [18], *Dempster-Shafer Theory* [190] and *Fuzzy reasoning* [130].

[5]Recall the definition for inconsistency in section 1.2.2.1, where an inconsistency was also defined as a logical contradiction: that is, one can derive from a formal system that a statement is both true and false.

2.2.4 Properties of Formal Systems

Three important properties of formal systems are *consistency*, *completeness* and *decidability.* As hinted in the previous section, (logic) formal systems are *consistent* if it is not possible to derive both a formula and its negation.

Formal systems are (syntactically) complete if for every well-formed expression A either A or $\neg A$ is a theorem. That is, all elements of a formal language can also be derived using the inference rules and axioms. A logic calculus is *semantically complete* with respect to an interpretation I if all well-formed formulas of the formal system that are true in I are also theorems (i.e., are *models*[6] (see section 2.2.2)) [204].

A formal system is *decidable* if there exists an algorithm that can calculate a characteristic function χ which can determine whether an arbitrary expression is a theorem or not. Such a function is defined as:

$$\chi_{\mathcal{L}}(w) = \begin{cases} 1 & \text{if } w \in \mathcal{L} \\ 0 & \text{if } w \notin \mathcal{L} \end{cases} \tag{1}$$

Formal systems are *semi-decidable* if there exists an algorithm which can recognize all theorems of formal system, but may not return an answer if the given expression is not a theorem (i.e., it may not be an effective procedure for checking that a formula is not a theorem). An example of a decidable calculus is *propositional logic. First-order predicate logic* (FOL) is an example of a semi-decidable formal system. As mentioned in section 2.1.1, not all languages are decidable. *Peano arithmetic*, for instance, is an example of an undecidable calculus – that is, a calculus for which a (single) algorithm for determining whether an arbitrary expression is a theorem

[6]Note that the use of the word *model* over the past sections has been used with two different definitions. To model a phenomenon is to construct a *formal theory* that describes and explains it. In a closely related sense, one models a system or structure that one plans to build, by writing a description of it. These are very different senses of *model* from that in model theory: the model of the phenomenon or the system is not a structure, but a *theory*, often in a formal language. UML, for instance, is a formal modeling language designed for just this purpose.

cannot be constructed [113].

2.3 Bayesian Reasoning & Learning

Bayesian probability theory is one of several generally accepted interpretations of the concept of probability and belongs to the category of evidential probabilities [18, 212]. Within the context of decision theory, it has been proven that Bayesian probability provides the only suitable basis for admissible decision rules [212]. In contrast to the *frequentist* perspective on probability theory, probabilities are not treated as frequencies of some phenomenon, but rather as a degree of *belief* about a phenomenon [18]. Such phenomena are typically depicted by random variables – i.e., variables whose values are subject to variations due to randomness. The state of belief can be interpreted as a *willingness to bet* on the occurrence of a particular (randomly occurring) phenomenon [40].

Bayesian probability is sometimes seen as an extension to propositional logic that enables reasoning with propositions whose truth value is uncertain [125]. This is in agreement with the definitions given in section 2.2.3.2, where Bayesian probability theory (and, in particular Bayesian inference) was mentioned to be an *inexact* reasoning method. Unlike in mathematical logic, where theorems are derived starting from a set of axioms using rules of inference, Bayesian inference is based on the principle of determining an updated probability distribution starting from a prior belief (expressed as a probability distribution) and a set of additional observations (i.e., new information that is received) relevant to the context.

2.3.1 Probability Basics

Before introducing Bayesian inference and Bayesian learning, necessary basic definitions and principles from probability theory are introduced. These include the notion of a probability space, conditional probability and random variables. It should be noted that the introduction to probability theory is kept very brief. Further details can be obtained from the numerous textbooks on probability theory.

2.3.1.1 Probability Space

Probability theory has to do with *experiments* that have distinct *outcomes* which occur with uncertainty (i.e., at random). An example of an experiment is the flipping of a two-sided coin, in which the outcomes are either *heads* or *tails*. Another example is to *pick a (randomly selected) student from a population of students* and determining whether the student is *enrolled in a Ph.D. program* or *not enrolled in a Ph.D. program*. The uncertainty associated with the outcome of an experiment is indicated by a *probability measure*, which is simply a real number between 0 and 1. A *probability space* defines the possible outcomes of an experiment and defines a probability measure.

Once an experiment is well-defined, the collection of all outcomes is called the *sample space* (or *outcome space*) and is denoted by the symbol Ω. Mathematically, a sample space is a set and the outcomes are the elements of the set – that is, a sample space with n outcomes is defined as $\Omega = \{\omega_1, \omega_2, ..., \omega_n\}$. For brevity, only finite sample spaces are considered in this introduction to probability theory.

A subset of the sample space is called an *event*. In the following, the convention will be used that events are denoted by capital letters from the beginning of the alphabet (e.g., A or B). A subset containing exactly one outcome is called an *elementary event*. Certain sets of events – i.e., certain sets of subsets of Ω – are known as σ-algebras $\mathcal{F}$. For finite sample spaces, $\mathcal{F} \subseteq 2^{\Omega}$, where 2^{Ω} is the *power set* (i.e., the set of all subsets including the empty set and the set itself). σ-algebras are used in constraining the possible sets over which probabilities can be defined. Probabilities are assigned to events and elementary events using a probability measure, which is a function $P : \mathcal{F} \to [0, 1]$ that defines a mapping from elements of $\mathcal{F}$ to a real number from the range 0 to 1.

Note that in order for a set of subsets of Ω to be a valid σ-algebra, the set $\mathcal{F}$ must fulfill certain conditions:

- $\mathcal{F}$ must contain the sample space (i.e., $\Omega \in \mathcal{F}$)
- $\mathcal{F}$ must be closed under complementary events: that is, if $A \in \mathcal{F}$, then $(\Omega \setminus A) \in \mathcal{F}$ must also hold, and as a consequence, $\emptyset \in \mathcal{F}$
- $\mathcal{F}$ must be closed under countable unions: for any elements $A_1, A_2, A_3, ...$ of $\mathcal{F}$, it must hold that $(\bigcup A_i) \in \mathcal{F}$ (i.e., their union must also be an element of the σ-algebra)

A valid probability measure P must also satisfy a number of conditions. These conditions are also known as *Kolmogorov's axioms of probability theory* [131]:

- For any ω_i, it must hold that $0 \leq P(\{\omega_i\}) \leq 1$
- $\sum P(\{\omega_i\}) = P(\Omega) = 1$ (i.e., the sum of the probability of all elementary events must equal one)
- P must satisfy the *countable additivity* property: for all sets of pairwise disjoint (independent) events A_i, the condition must hold that $P\left(\bigcup A_i\right) = \sum P(A_i)$

Once a sample space Ω has been constructed, and a valid probability measure P and σ-algebra $\mathcal{F}$ have been defined, a valid probability space has also been defined. Formally, a probability space is denoted by the triple $(\Omega, \mathcal{F}, P)$.

2.3.1.2 Conditional Probability

A *conditional probability* is the probability of an event, given that another event has occurred. Consider two events, A and B. As introduced previously, both A and B are sets of outcomes. Say that $B \cap A \neq \emptyset$: then sets A and B overlap. This is illustrated in figure 3 using a *Venn Diagram* [152]. Intuitively, if all outcomes are equally likely,

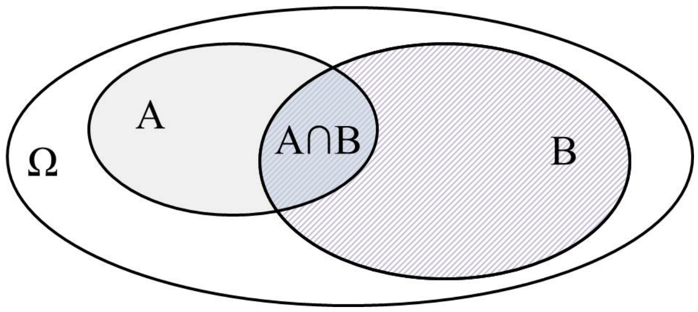


Figure 3: Venn diagram illustrating the intersection of two events A and B.

the probability of the outcome of the experiment being in A given that it is already known that the outcome is also in B is then the ratio of the outcomes in both A and B (the overlap), and those in B (since it is known that B has already occurred). Dividing by the total number of outcomes, the conditional probability can be defined as a ratio of two probabilities as in equation 2:

$$P(A \mid B) = \frac{|A \cap B|}{|B|} = \frac{|A \cap B| \ / \ |\Omega|}{|B| \ / \ |\Omega|} = \frac{P(A \cap B)}{P(B)} \tag{2}$$

Note that, similar to the convention used in section 2.1.1.1, $|A|$ denotes the size of a set A (i.e., the number of elements). Also note that the condition must hold that $P(B) \neq 0$. Furthermore, while derived using the assumption that all outcomes are equally likely, the rightmost term of equation 2 is also valid for the general case of outcomes possibly not being equi-probable. Finally, note that, in the following, the expressions $P(A_i \cap A_j)$ and $P(A_i, A_j)$, both denoting the probability of events A_i *and* A_j occurring, are used interchangeably.

Events may also be *independent*. Assume two other events C and D with $P(C) \neq 0$ and $P(D) \neq 0$. If $P(C \mid D) = P(C)$, then C is said to be independent of D. This is a result from $C \cap D = \emptyset$. Note that independence is symmetric: i.e., $P(D \mid C) = P(D)$.

Conditional independence is defined similarly, with the addition that two events are only independent when given a third event. Say that A and B are independent if an event F has already occurred. Then, $P(A \mid F) = P(A \mid B, F)$ and it is said that A and B are conditionally independent given F. However, as before, $P(A \mid B) \neq P(A)$.

An interesting and very useful rule to computer the *joint probability* of a set of events A_i (i.e., the probability that all events A_i occur simultaneously) can be derived from the definition of conditional independence: the *chain* or *multiplication rule* [152]. For n events, the chain rule takes the following form:

$$P(A_1, A_2, ..., A_n) = P(A_1)P(A_2 \mid A_1)P(A_3 \mid A_1, A_2)...P(A_n \mid A_1, A_2, ..., A_{n-1}) \quad (3)$$

This rule can easily be proven by iteratively simplifying the right hand side using the definition of conditional probability given in equation 2 (for the full proof see the related literature, e.g., [152]).

A similar and related rule is the *law of total probability* [152]. Suppose one is given n mutually exclusive and exhaustive events $E_1, E_2, ..., E_n$ such that $E_i \cap E_j = \emptyset$ for $i \neq j$ and $\bigcup_i E_i = \Omega$. Then the law of total probability states that for any other event F

$$P(F) = \sum_{i=1}^{n} P(F \cap E_i) = \sum_{i=1}^{n} P(F \mid E_i)P(E_i)\ . \quad (4)$$

2.3.1.3 Random Variables

The last major concept introduced in this brief overview of probability theory is that of a *random variable.* Random variables are mathematical variables whose values vary due to randomness. Mathematically, a random variable is a function on the sample space Ω that takes on a value from a *target space* E[7]:

$$X : \Omega \longrightarrow E \quad (5)$$

A *measurable space* is a pair (E, ϵ), where E is some space of values and ϵ a σ-algebra on E. For a given probability space $(\Omega, \mathcal{F}, P)$, a random variable X defined according to equation 5 is said to be $(\mathcal{F}, \epsilon)$-*measurable* and (E, ϵ)-*valued.* Note that, for brevity,

[7]Note that E is not an event.

only discrete and finite target spaces are considered[8]. There, the σ-algebra is, similar to before, defined as $\epsilon \subseteq 2^E$.

Random variables can be defined through their preimage. For any possible value, or combinations of values $x \in \epsilon$, this preimage is, necessarily, an event over which a probability is defined: $X^{-1}(x) \in \mathcal{F}$. Therefore, through the definition of the preimage of a random variable X, the mapping of the random variable can be defined:

$$X^{-1}(x) = \{\omega_i \in \Omega \mid X(\omega_i) = x\} \tag{6}$$

For a random variable, $X = x$ is used to denote the set of all elements $\omega_i \in \Omega$ that X maps to the value x. Note that since random variables are defined over the sample space, a mapping must be defined for every possible outcome of an experiment – i.e, for all elements of Ω. Since a random variable typically has a multi-valued target space and, hence, maps from more than one possible event, the expression $p(X)$ is used to indicate the *probability distribution* of X. $P(X = x)$ is then the probability of $X = x$. The *joint probability distribution* is a probability function on the Cartesian product of the spaces of the random variables. The joint probability distribution of two random variable X and Y is denoted by $p(X, Y)$.

Given a joint probability distribution, the law of total probability (see equation 4) can be applied to obtain the probability distribution of any one random variable by summing over all values of the other variables. This is known as the *marginal probability* (distribution). Say random variables X and Y are given. The marginal probability for a discrete random variable X for $X = x$ is then:

$$P(X = x) = \sum_i P(X = x, Y = y_i) \tag{7}$$

[8]In most textbooks, random variables are often introduced with numeric target spaces which are either continuous or discrete (i.e., finite or countable). For instance, for continous random variables, one often encounters the case of $E = \mathbb{R}$, in which case ϵ is typically chosen to be the Borel σ-algebra [18]. While numeric target spaces are very commonplace, target spaces can also be sets of labels, in which case the random variables are often referred to as *categorical*.

The definition of marginal probability distributions for continuous random variables is analogous. There, the difference is that, instead of summing over all elements, integration along the dimension(s) to be marginalized is performed.

Throughout this dissertation, random variables will typically (unless specifically stated) be denoted by capital letters from the end of the alphabet, usually X or Y. Note that the definition of conditional probability is analogous to that introduced for events.

2.3.1.4 Moments of Random Variables

Probability distributions are often characterized by specific quantitative measures. These measures are known as the *moments*. The *zeroth moment* is the total probability (i.e., 1), the first moment is the *expected value* or *mean*, and the second moment is the *variance*. Here, the latter two are briefly introduced.

If the target space of a random variable X is a subset of the real numbers (e.g., the natural numbers), the expected value of X is given by:

$$\mathbb{E}\left[X\right] = \sum_{i=1}^{n} x_i p_i \tag{8}$$

Here, x_i represents the value in the target space and p_i represents the probability $P(X = x_i)$. The expected value is the mean or average value of X that one would expect if a large number of experiments were performed repeatedly. Note that if the target space is infinite (but still countable) then $n = \infty$.

For a target space of real numbers, the expected value of a random variable can be determined using its *probability density function* (pdf)[9] $f(x)$. The integral of the probability density function between limits a and b is the probability of the random variable falling within the range $[a, b]$. Therefore, the expected value for continuous

[9]Probability density functions are commonly used to describe probability distributions (see the related literature (e.g., [18]) for a list of probability density functions for common distributions). A well-known example is the function that describes the "bell"-shaped curve of the normal distribution. Such functions describe the relative likelihood for a random variable to take on a given value.

random variables can be defined in a similar way to that of discrete random variables:

$$\mathbb{E}\left[X\right] = \int_{-\infty}^{\infty} x f(x) \, \mathrm{d}x \tag{9}$$

The second moment (or variance) of a random variable measures the spread of the associated distribution. For example, a variance of zero indicates that all values are identical. A (in relative terms) small variance indicates that the outcome will be very close to the mean, whereas large variances indicate a large spread. The variance is often used as a parameter to describe probability distributions (at least in part). Variance is the square root of the *standard deviation* σ, which indicates deviations from the mean. Hence, variance is formally defined as the expected value of the squared deviation from the mean:

$$\mathrm{Var}\left(X\right) = \sigma^2 = \mathbb{E}\left[(X-\mu)^2\right] = \mathbb{E}\left[X^2\right] - \left(\mathbb{E}\left[X\right]\right)^2 \tag{10}$$

Note that in equation 10 the symbol μ was used to indicate the mean value – i.e., the expected value of the random variable. For discrete random variables, equation 10 simplifies to:

$$\mathrm{Var}\left(X\right) = \sum_{i=1}^{n} p_i \left(x_i - \mu\right)^2 = \sum_{i=1}^{n} p_i x_i^2 - \mu^2 \tag{11}$$

For the case of continuous random variables with probability density function $f(x)$, the variance is given by:

$$\mathrm{Var}\left(X\right) = \int \left(x-\mu\right)^2 f(x) \, \mathrm{d}x = \int x^2 f(x) \, \mathrm{d}x - \mu^2 \tag{12}$$

Like before, μ refers to the expected value. Related to the concept of variance is also that of *covariance*, which is a measure of how much two random variables change together.

2.3.2 Bayesian Inference

Bayesian inference provides a probabilistic approach to inference [18]. At its core is Bayes' Theorem (see equation 13) [18]. Bayes' Theorem states that, given two events

A and B such that $P(A) \neq 0$ and $P(B) \neq 0$, the probability of event A occurring, given that B has been observed can be determined by calculating:

$$P(A \mid B) = \frac{P(A)\ P(B \mid A)}{P(B)} \tag{13}$$

Note that equation 13 follows directly from the definition of conditional probability given in equation 2. Within the context of a Bayesian interpretation of probability, the event that has already occurred (here: B) is also referred to as the *evidence* or *information* for the conditioned event.

Bayes theorem is the cornerstone of Bayesian machine learning methods because it provides a means of calculating the *posterior probability* $P(X = x \mid Y_E = y_E)$ of an event $X = x$ given some observed evidence $Y_E = y_E$ from the *prior probability* $P(X = x)$, and the conditional probability $P(Y_E = y_E \mid X = x)$ and $P(Y_E = y_E)$. In machine learning – specifically in the subfield of concept learning – the most likely event H_{MAP}, and hence the value x_{MAP} of a random variable X that maximizes the probability is often of interest. The target space of X can be either continuous or discrete. In the case of a discrete space each element in the target space of X represents a particular concept (e.g., in the form of a categorical label or proposition). In some applications of inexact reasoning, the intent of using Bayesian inference is to identify the most likely event called the *maximum aposteriori* (MAP) event[10] [149].

$$H_{MAP} \equiv \operatorname*{argmax}_{X=x} P(X = x \mid Y_E = y_E) \tag{14}$$

To determine the MAP event, a naïve algorithm would simply compute the probability for each possible $X = x$ and then select the x for which the probability $P(X = x \mid Y_E = y_E)$ is maximal. Depending on whether X is discrete or continuous, and on the size of the target space of X, this algorithm can be computationally very expensive. However, for small target spaces – particularly small (finite) discrete

[10]In the related literature on machine learning, this maximum aposteriori event is also often referred to as the MAP *hypothesis*, selected from a space of possible hypotheses.

spaces – the computation is commonly (at least in most practical cases) tractable.

2.3.3 Bayesian Networks

Bayesian inference is fairly simple if the number of random variables and their target spaces are very small. However, as the number of variables increases, the number of terms that need to be made available and computed with rises exponentially. Bayesian networks [169] reduce the complexity in such cases. If (conditional) independence assumptions can be made explicit apriori, then Bayesian networks are compact representations of the joint probability distributions over the set of random variables they contains, and, by exploiting some of their properties, allow for efficient inference.

2.3.3.1 Difficulties in Solving Large Instances

Consider the case, where Bayes' theorem (see equation 13) is applied for computing the posterior probability distribution for a random variable X given the occurrence of n events $Y_1 = y_1, Y_2 = y_2, ..., Y_n = y_n$. Using the chain rule from equation 3, the following can be shown to hold true:

$$p(X \mid Y_1 = y_1, Y_2 = y_2, ..., Y_n = y_n) = \frac{p(X, Y_1 = y_1, Y_2 = y_2, ..., Y_n = y_n)}{P(Y_1 = y_1, Y_2 = y_2, ..., Y_n = y_n)}$$

Both the numerator and denominator of this equation are available given the joint probability distribution over the random variables (where the denominator is determined using marginalization (see equation 7)). However, in practice this is typically not the case. Primarily, this is due to the associated challenge of having to capture a very large number of probabilities. To illustrate this, let X and all Y_i be *binary random variables* – that is, discrete random variables with 2 values. The number of possible combinations of values is, in this case, $2^{(n+1)}$. For a moderate number of random variables – say 16 – the number of possible combinations and, therefore, the number of individual probabilities required to fully define the joint probability distribution over the random variables, is $65,536$. However, this assumes the absence of (conditional) independence among random variables, opening up the possibility that some (if not many) of the $65,536$ entries are redundant.

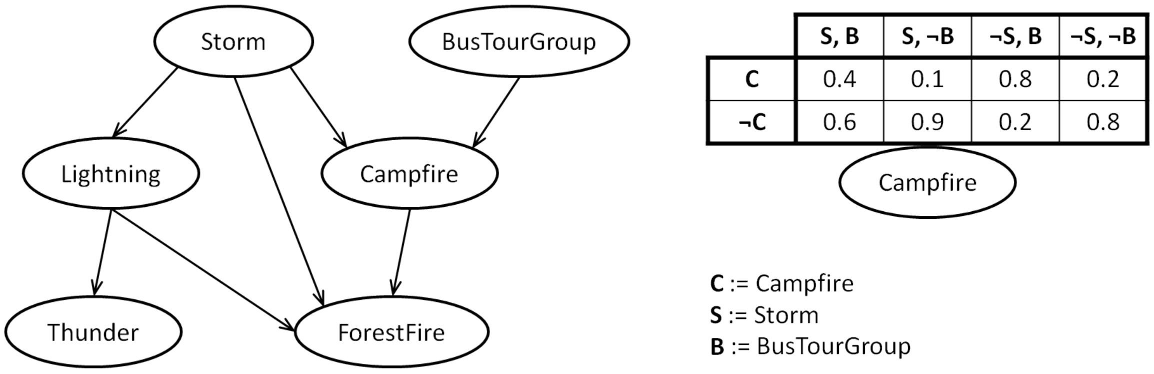
2.3.3.2 Bayesian Networks: a Definition

Bayesian belief networks – or simply *Bayesian networks* – address both the problem of representing the joint probability distribution over a large number of random variables and performing inference with these variables [170, 153]. Formally, a Bayesian belief network is a tuple $(\mathbb{G}, P)$, where P is a probability function over a set of random variables $V = \{X_1, X_2, ...\}$, and $\mathbb{G} = (V, E)$ is a *directed acyclic graph* (DAG) (i.e., a graph with directed edges where following the edges in the indicated direction from any starting vertex will never result in reaching the same starting vertex) whose vertices are variables in V and whose directed edges are defined as $E \subseteq V \times V$ (i.e., a subset of the set of all pairs of elements from V, where the first element indicates the source vertex, and the second the target vertex). In addition, $(\mathbb{G}, P)$ must satisfy the *Markov condition*: for each variable $X_i \in V$, X is conditionally independent of the set of all its non-descendants given values for the set of all its parents $pa(X)$ [152].

The result of the Markov condition is that the joint probability distribution represented by a Bayesian belief network is equal to the product of the conditional distributions of all random variables (represented by vertices) given values of the random variables represented by *parent vertices*[11] whenever these conditional distributions exist [170]. Because of the acyclic nature of the graph, the set of parents is a subset of the set of non-descendants. This reduces the set of unknowns to only the conditional distributions of the random variables X_i in V given values of their parents $pa(X_i)$ in the Bayesian network. These distributions are known as the *parameters* of a Bayesian network and are typically captured in *conditional probability tables.*

Figure 4 illustrates an example Bayesian network adopted from [149]. It represents

[11]Parent vertices in a graph are those vertices that have a directed edge pointing towards the child vertex.



	S, B	**S, ¬B**	**¬S, B**	**¬S, ¬B**
C	0.4	0.1	0.8	0.2
¬C	0.6	0.9	0.2	0.8

Figure 4: An example of a Bayesian network (adapted from [149]) showing conditional independence assumptions, and the conditional probability table for the vertex *Campfire*.

the joint probability distribution over the binary random variables *Storm*, *Lightning*, *Thunder*, *ForestFire*, *Campfire*, and *BusTourGroup*. To name but one example, the network represents the assertion that *Campfire* is conditionally independent of its nondescendants *Lightning* and *Thunder* given values for its immediate parents *Storm* and *BusTourGroup*. This means that once the values for *Storm* and *BusTourGroup* are *known* (through observation), the variables *Lightning* and *Thunder* provide no additional information about *Campfire*. Figure 4 also illustrates an example of a conditional probability table. Entries are read in the following manner: for instance, the entry on the upper left (C and S, B) is the entry representing $P(\mathit{Campfire} = \mathit{true} \mid \mathit{Storm} = \mathit{true}, \mathit{BusTourGroup} = \mathit{true}) = 0.4$.

A Bayesian network is typically developed by first creating a DAG $\mathbb{G}$ such that it is believed that every random variable $X_i \in V$ satisfies the Markov condition locally. Typically, this is done by creating a *causal* graph – i.e., a graph where the directed edges indicate causes. This is also known as defining the *structure* of the Bayesian network. The conditional probability distributions of each variable given values of their parents (the parameters) are then elicited. If the joint probability distribution of all random variables $X_i \in V$ is then defined as the product of these conditional distributions, then the tuple $(\mathbb{G}, P)$ is a Bayesian network.

2.3.4 Inference in Bayesian Networks

The Markov condition states that each variable is (locally) conditionally independent of its non-descendants given its parent variables. This condition, along with information about the conditional dependence significantly reduces the number of terms required to fully define the joint probability distribution represented by a Bayesian belief network. To determine the joint probability, the product of the conditional probabilities of all random variables $X_i \in V$ given values of their parents $pa(X_i)$ (whenever these conditional distributions exist) can be determined [152] (see equation 15):

$$P(X_1 = x_1, X_2 = x_2, ...) = \prod_i P(X_i = x_i \mid pa(X_i)) \tag{15}$$

This enables the computation of a conditioned posterior probability distribution using Bayes' theorem (equation 13) for any combination of random variables represented by a given Bayesian network. For instance, referring back to the example network from figure 4, say that one would like to *infer* the probability of a *ForestFire* (F) occurring given that *Lightning* (L) is observed. In this case, the application of Bayes' theorem and the reverse chain rule leads to:

$$P(F \mid L) = \frac{P(L \mid F)P(F)}{P(L)} = \frac{P(L, F)}{P(L)}$$

Note that F and L were used to indicate the events *ForestFire* = *true* and *Lightning* = *true*, respectively. The joint probabilities in the numerator and denominator can be determined through marginalization. For instance, to determine $P(L, F)$, the following sum must be determined (see equation 7):

$$P(L, F) = \sum_{x \in \{true, false\}} P(L, F, C = x, B = x, T = x, S = x)$$

Note that C (*Campfire*), B (*BusTourGroup*), T (*Thunder*) and S (*Storm*) denote the remaining random variables in the example Bayesian network. The terms of the sum can be computed using equation 15. The inferred probability of a *ForestFire* occurring given that *Lightning* is observed is *consistent with the beliefs over the network parameters* given the assumptions made about independence, since only the axioms and theorems of probability theory were applied.

For large Bayesian networks, similar computations can be computationally very expensive because of the large number of terms involved. Indeed, it was shown that, in general, *exact* inference for an arbitrary Bayesian network is NP-hard [34]. Therefore, in practice, this naïve algorithm (sometimes referred to as *naïve enumeration*) is commonly not used. Research related to performing inference in Bayesian networks has lead to a number of highly efficient algorithms for performing inference in Bayesian networks. Most of these make use of pre-computations and properties of the graph associated with a Bayesian network. Well-known are the *variable elimination* [234] algorithm, and its generalization, the *junction tree* (or *clique tree*) algorithm [134]. Besides *exact* inference algorithms, there are also *inexact* inference algorithms which sacrifice precision to gain efficiency (e.g., Monte-Carlo based [39]).

2.3.5 Learning Bayesian Network Parameters

In the previous section it was assumed that a Bayesian network parameter is specified through a conditional probability table that is filled out by a human. In this section, basic principles of *Bayesian learning* for Bayesian network parameters are introduced. In other words, methods for *updating* a *prior belief* on a network parameter using data are outlined. It is shown how beliefs can be updated *automatically* using a series of observations of the random variables in the corresponding Bayesian network.

To illustrate how Bayesian network parameters can be learned, let $(\mathbb{G}, P)$ be a simple Bayesian network with one binary random variable $\mathbf{X}$ – i.e., represented by a DAG with $V = \{\mathbf{X}\}$ and $E = \emptyset$. Therefore, the only network parameter of the Bayesian network is the probability distribution over $\mathbf{X}$, which is defined by

$p(\mathbf{X} \mid pa(\mathbf{X}))$. In this case, $pa(\mathbf{X}) = \emptyset$, since no other random variables are a part of the network. Assume that a *data set* $\mathbf{D}$ is a vector of *data cases* D_i, where each data case is an observation of $\mathbf{X}$ (i.e., each D_i is either X or $\neg X$, where X represents the event associated with $\mathbf{X} = \textit{true}$ and $\neg X$ the event associated with $\mathbf{X} = false$). Let this data set have n entries and be defined as:

$$\mathbf{D} = \{D_1 = X, D_2 = \neg X, D_3 = \neg X, ..., D_n = X\}$$

Let $P(X) = \theta$ and $P(\neg X) = 1 - P(X)$. θ can now be *learned* from the data set by solving the regression problem *"find the θ that best fits the given data"*.

2.3.5.1 Maximum Likelihood Estimation

One way of looking at this problem is to answer the question: *"which value of θ maximizes $P(\mathbf{D} \mid \theta)$?"* In the literature, this is referred to as *Maximum Likelihood Estimation* (MLE) [152]. MLE entails finding the θ^* that maximizes the *likelihood* of θ given $\mathbf{D}$, i.e., for $L(\theta \mid \mathbf{D}) = P(\mathbf{D} \mid \theta)$ find $L(\theta^* \mid \mathbf{D}) = \sup_\theta L(\theta \mid \mathbf{D})$. If independence of the data cases is assumed (which is, for most problems, a valid assumption [149]), $P(\mathbf{D} \mid \theta)$ is simply $P(\mathbf{D} \mid \theta) = \prod_{i=1}^{n} P(D_i \mid \theta) = \theta^{n_X}(1-\theta)^{n_{\neg X}}$, where n_X is the number of data cases where $D_i = X$ and $n_{\neg X}$ the number of data cases where $D_i = \neg X$. To find θ^*, the derivative with respect to θ of the logarithm of the likelihood is set to zero, leading to the intuitive result $\theta^* = n_X/(n_X + n_{\neg X})$ (i.e., the *relative frequency* of X in the dataset).

Given that all related assumptions hold, the MLE of θ will (typically) reach the *true* value as $n \to \infty$. However, especially for small sets of data, the MLE will not reflect what one would *expect* to observe – i.e., it is not a good representation of one's beliefs. A good example of this is a limited number of trials in an experiment where a coin is tossed. In addition, for some cases, the maximum likelihood estimate can lead to a *biased* estimator. This is the case if the true value for θ and the value determined MLE differ. Specifically, this is the case for Gaussian distributions, where, when estimating the mean, the true value and the mean determined using MLE differ. However, if the data set is large, this difference is negligible [149].

Another issue with the MLE is that it can produce unintuitive results for *unlikely* events. In such cases, it may be that, even for a large data set, a particular event is never observed and the probability of the event occurring is determined to be 0. Therefore, a better approach is to determine the θ that maximizes $P(\theta \mid \mathbf{D})$ rather than the likelihood of θ. This is known as *Bayesian estimation*.

2.3.5.2 Bayesian Estimation

As mentioned in the previous section, estimating the probability distributions over Bayesian network parameters using MLE can be a useful if none of the events are very unlikely and if the data set is very large. Again, assume the same Bayesian network as in the previous section. MLE is used in determining a value for θ that maximizes the likelihood of θ given the data cases provided. From a Bayesian perspective this makes little sense, since it is not θ that should be a fixed value, but the data set is fixed. Therefore, it can be concluded that a better way is to determine the θ that maximizes $P(\theta \mid \mathbf{D})$. However, this requires imposing a distribution over the data and θ.

Let θ now be a random variable (and no longer a fixed value) – more specifically, let θ be a representation of our belief on the value of the corresponding network parameter $P(X)$ (which is any real value from the range [0, 1]). Bayesian estimation dictates that this *prior* belief on θ should now be *updated* (using Bayes' theorem) with the provided data cases to form a *posterior belief* $p(\theta \mid \mathbf{D})$. Given that θ is no longer fixed, but a random variable, the belief on the Bayesian network parameter should now be rewritten as:

$$P(X \mid \mathbf{D}) = \int_0^1 P(X, \theta = f \mid \mathbf{D})df = \int_0^1 P(X \mid \mathbf{D}, \theta = f)P(\theta = f \mid \mathbf{D})df \qquad (16)$$

Equation 16 can be simplified by incorporating the following observations: for one, observing the true value of θ, any additional observations about $\mathbf{X}$ should not change the value of θ. Therefore, $P(X \mid \mathbf{D}, \theta = f) = P(X \mid \theta = f)$. Also note that, given that θ is a representation of our belief on $P(X)$, knowing the true value for θ results in the additional simplification $P(X \mid \theta = f) = f$. The only unknown term remaining then is $p(\theta \mid \mathbf{D})$. However, this term can be computed through application of Bayes' theorem: $p(\theta \mid \mathbf{D}) \propto p(\theta)P(\mathbf{D} \mid \theta)$. Under the assumption that all data cases are independent, $P(\mathbf{D} \mid \theta = f) = f^{n_X}(1-f)^{n_{\neg X}}$ (see derivation of MLE from previous section). This leads to the following expression:

$$P(X \mid \mathbf{D}) = c \int_0^1 f P(\theta = f) f^{n_X}(1-f)^{n_{\neg X}} df \tag{17}$$

Here, c is a normalization constant (a result from applying Bayes' theorem). Note that the likelihood is binomial. To find a closed form expression for equation 17, the (continuous) distribution over θ should be from a conjugate family of binomial distributions. For this purpose, a *Beta* distribution $B(\alpha_X, \alpha_{\neg X})$ is chosen, reflecting our *prior* belief on the value of the Bayesian network parameter $P(\mathbf{X})$. It can be shown that equation 17 then reduces to:

$$P(X \mid \mathbf{D}) = \frac{n_X + \alpha_X}{n_X + \alpha_X + n_{\neg X} + \alpha_{\neg X}} = \frac{n_X + \alpha_X}{n + \alpha} \tag{18}$$

The use of Beta distributions for capturing priors on Bayesian network parameters has the nice side effect that Beta distributions are generally considered to be a good choice for eliciting beliefs on binary events [152]. Also note that depending on the choice of α_X and $\alpha_{\neg X}$ (which can both be > 1), the prior *can* prevail very strongly, even if large numbers of data cases are considered[12]. This allows one to express a degree of certainty in an event in relation to a population size.

[12]The sum of the parameters of the Beta distribution are also known as the *prior sample size*.

2.3.5.3 *Learning Multi-Valued, Discrete Random Variables*

A similar result to that from equation 18 can be reached for multi-valued random variables. Let $\mathbf{X}$ now be a random variable with r values. Furthermore, let $\mathbf{D} = \{D_1 = x_1, D_2 = x_4, D_3 = x_2, ..., D_n = x_1\}$, where D_i takes on any value for which $\mathbf{X}$ is now defined. Then, θ_i is a random variable denoting the uncertain value of $P(\mathbf{X} = x_i \mid \mathbf{D})$. Let $\theta = (\theta_1, \theta_2, ..., \theta_r)$. The likelihood of $P(\mathbf{D} \mid \theta)$ is now no longer binomial, but *multinomial*.

To find a closed form expression for $P(X \mid \mathbf{D})$, a distribution on θ from a conjugate family of multinomial distributions should be chosen. One such distribution is the *Dirichlet* distribution $Dir(\alpha_1, \alpha_2, ..., \alpha_r)$, which reduces to the Beta distribution for $r = 2$. It can be shown [152] that the closed form reduces to:

$$P(X = x_i \mid \mathbf{D}) = \frac{n_i + \alpha_i}{n + \alpha} \tag{19}$$

Here, $\alpha = \sum_i \alpha_i$ and n_i is the count of the data cases in $\mathbf{D}$ with value x_i. Note that, under the given assumptions, the Bayesian network parameter is simply defined by the numerical mean of the ith dimension of a distribution $Dir(\alpha_1+n_1, \alpha_2+n_2, ..., \alpha_r+n_r)$. For brevity, only the end result is shown here. However, the interested reader will find the full derivation in [152].

2.3.5.4 *Learning Parameters of General Bayesian Networks with Discrete Random Variables*

The results from the previous section can be expanded to the general case of a Bayesian network with m discrete random variables X_i. Let each data case D_i now be a vector of m values, one for each of the m random variables. This leads to $\mathbf{D}$ now taking the shape of a *table*. Furthermore, let $\theta_{ijk} = P(X_i = j \mid pa(X_i) = k, \mathbf{D})$, where $pa(X_i) = k$ represents a particular *configuration* of values of the parent variables in the network (one permutation of the values of the parent random variables).

The derivation of the closed form is similar to that from the case of a single multi-valued discrete random variable. In addition to the assumptions made previously (e.g., independence of data cases), in the derivation it is assumed that the $\theta_{i..}$s are *globally* independent. In addition, it is assumed that the $\theta_{i.k}$ are independent – i.e., that there is a local independence with respect to parent configurations. Implied from this is an independence among Bayesian network parameters. Using these assumptions, it can be shown [152] that the following closed form expression exists:

$$P(X_i = j \mid pa(X_i) = k, \mathbf{D}) = \frac{n_{ijk} + \alpha_{ijk}}{\sum_j (n_{ijk} + \alpha_{ijk})} \tag{20}$$

Note that a *very* large number of data cases must be considered in order for sensible estimations of the Bayesian network parameters to be possible.

2.4 Summary

In this chapter, an overview of foundational concepts of formal languages, and exact and inexact reasoning methods is presented. Three distinct topics are introduced: fundamentals of formal languages (and formal modeling languages), automated reasoning in formal systems, and basic concepts in Bayesian probability theory.

In the first part, fundamentals of formal methods, languages and models are introduced. It is established that there is no principle difference in the definition of formal languages and formal modeling languages other than the syntax of a formal modeling language generally being iconic and visual in nature. Formal languages are purely syntactic in nature and are defined by utilizing meta-languages. The definition of formal semantics allows meaning to be assigned to utterances of formal languages. Formal semantics are defined by a domain of discourse, and a mapping from expressions to this domain of discourse. It is shown that in order to formally define semantics, a syntactic representation of formal semantics is required (e.g., for the domain of discourse). However, it is also explained that the degree of formality of this representation depends on the intended audience. Finally, graphs and graph transformations are shown to be suitable formalisms for defining the syntax and semantics of modeling languages.

The second part introduces the concept of a formal system and automated reasoning (automated proofs). Formal systems are composed of a set of axioms and inference rules, through which well-formed expressions and theorems can be derived. It is noted that most formal systems define the set of well-formed expression and theorems as equal. A proof checks whether a given expression is a theorem of the language. The interplay between the interpretation of expressions using concepts from model theory and automated proof theory are also outlined. In model theory, the terms that occur in an expression are interpreted as mathematical structures (e.g., groups, fields or graphs), and fixed compositional interpretation rules determine how the truth value of the expression can be derived from the interpretation of its terms. Model-based deduction techniques use algorithms which try to systematically test all valuations of an expression to determine its satisfiability. Deductive proofs were shown to always result in logically correct conclusions (given the the underlying formal system is complete and consistent). However, it was also shown that for many languages, determining whether an arbitrary expression is a theorem is undecidable.

In the third part of the chapter, a brief review of basic concepts from Bayesian probability theory is given. Among the concepts presented are Bayesian networks, through which joint probability distributions can be represented in a compact fashion, and probabilistic inference performed efficiently. Bayesian networks are represented by acyclic graphs with random variables representing nodes, and directed edges between nodes representing influence relationships. This graph is known as the structure of a Bayesian network. In addition to the structure, a Bayesian network is defined by its parameters, which are defined by (conditional) probability distributions. Towards the end (in section 2.3.5), core aspects of learning such network parameters from data are presented.

CHAPTER III

RELATED WORK ON INCONSISTENCY MANAGEMENT

In this chapter, results from conducting a review of the related literature in inconsistency management are presented. This includes the introduction of existing frameworks, and notable approaches to inconsistency identification and semantic overlap detection from the related literature on software and systems engineering research. Methods from closely related fields, such as database and information systems are briefly reviewed also. The presented methods for inconsistency identification and semantic overlap detection are based on concepts introduced in chapter 2.

The primary purpose of this chapter is to provide an overview of the research conducted in the field of inconsistency management, with a focus on inconsistency identification and semantic overlap detection. Closely related approaches – e.g., those concerned with model integration and transformation – and their relevance to inconsistency identification is briefly discussed as well.

3.1 Overview of Related Work on Inconsistency Management

Identifying and resolving inconsistencies in formal models (i.e., *inconsistency management*) is a well-studied subject in the domain of software engineering. However, *automating* the identification (and resolution) of inconsistencies and semantic overlap remains an open challenge. This is particularly the case for application scenarios where multiple, disparate and incomplete models of a (software or physical) system are considered. Research in the domain of Model-Based Systems Engineering is especially lacking, where such modeling scenarios are frequent.

In the following, a review of the related literature on managing inconsistencies in formal models is presented to reflect the current state of research in inconsistency management. The review is meant to support the argumentation for the identified research gap presented in section 1.2.4. Included is notable research from the domains of software engineering and MBSE (and related domains such as mechatronics).

The earliest work on inconsistency management is from the domain of software engineering. Finkelstein is often credited with having coined the term *inconsistency management*, particularly based on his early work related to managing inconsistencies in modeling environments where multiple views are employed to express various concerns [69]. Since then, the interest in model-based development in general, and the problem of model integration has grown tremendously. Particularly in software engineering research, the maturity and general acceptance of (semi-)formal modeling languages, such as the *Unified Modeling Language* (UML) [166] has spawned a variety of related research efforts. This has led to a number of researchers investigating a variety of methods for managing inconsistencies.

3.2 Existing Inconsistency Management Frameworks

The first published conceptual frameworks related specifically to what has later become known as inconsistency management [157, 158] were developed by Finkelstein [69, 71, 70]. In what he has named the *Viewpoint-Oriented Systems Engineering* (VOSE) framework, the core principle is the use of viewpoints to partition the system specification (e.g., functional hierarchy and system block diagram), development method (e.g., "top-down" and "bottom-up") and formal representations used to express the system specifications [69]. Finkelstein defines viewpoints as compositions of several components, among which are the representational style, specification, work plan, and work record. The representational style is, essentially, the representation language definition and defines objects and relations that can be used in the specification. The work plan is defined to contain *assembly actions*, *check actions* and *guide actions*. Assembly actions are actions to be used by a developer to build a specification (e.g., to ensure well-formedness across the different views). Check actions are defined as actions (in the form of a rule) to be used to identify any inconsistencies in the specification. These actions are decomposed into *in-viewpoint* and *inter-viewpoint checks*. Inter-viewpoint checks are required for cases where there is an *overlap* between two viewpoints. Once an inconsistency has been discovered, guide actions provide the developer with guidance on what is to be done as a result. Guide actions also suggest when to perform which particular check actions. The approach taken by Finkelstein differentiates itself from others from the same time period in that a single representation scheme or common data model is not required (but it does assume translation between viewpoints is possible and defined). Finkelstein *et al.* refine the approach in [70]. In what the authors refer to as *interference management*, nine distinct activities are identified, the most important of which are: *overlap identification*, *consistency relation construction*, *inconsistency detection*, and *inconsistency resolution*.

Nuseibeh and Easterbrook build on the ideas of Finkelstein *et al.* by developing a concrete inconsistency management framework in [158]. The framework defines several activities related to the identification and resolution of inconsistencies. The core idea is the definition of *consistency rules* that are applied when monitoring for inconsistencies. Consistency rules are relationships between descriptions of a system that should hold. If the relationship is violated, an inconsistency exists. Once an inconsistency has been *detected*, it is *diagnosed* (i.e., *located*), its *cause identified*, and the inconsistency is *classified*. Thereafter, the inconsistency is *handled*, which may result in the *resolution* (or "fixing") of the inconsistency, or in *tolerating* it (e.g., if the cost of fixing exceeds the benefits). Tolerating can include *ignoring*, *circumventing* and *ameliorating* the inconsistency. A final step in the process is the

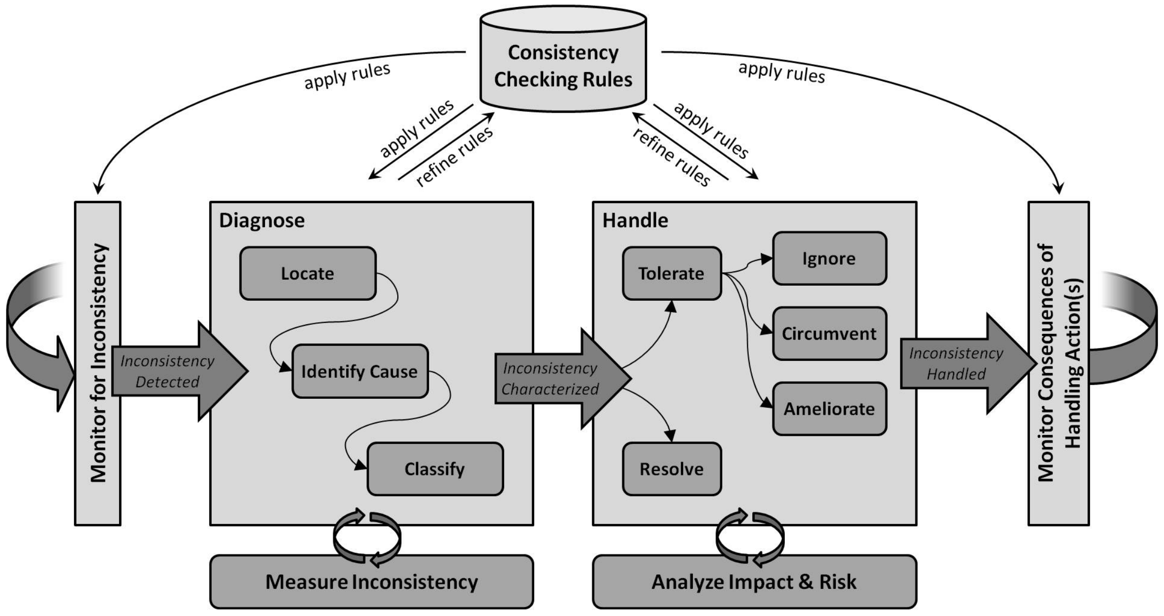


Figure 5: Framework for inconsistency management proposed by Nuseibeh and Easterbrook and introduced in [158].

monitoring of the consequences of a particular handling action. The framework also defines a *measuring activity*, which is used to measure the degree of inconsistency of a software specification, as well as an analysis of the impact and risk of a particular handling action. It is assumed that the set of consistency rules is (manually) extended over time. The existence of explicit relationships among different artifacts (i.e., the application of the consistency rules, and hence the definition of overlap relations) is assumed to be defined manually. Figure 5 illustrates and summarizes this framework.

Spanoudakis and Zisman merge the already introduced work by Finkelstein *et al.* [70] with that of Nuseibeh *et al.* [158, 157], and propose a general framework for inconsistency management in software engineering in [200]. Six distinct activities make up the framework: *detection of overlap*, *detection of inconsistencies*, *diagnosis of inconsistencies*, *handling of inconsistencies*, *tracking of inconsistencies* and *specification and application of an inconsistency management policy.* Several of the previously discussed activities have been either merged or sub-divided: for example, the *consistency relation construction* mentioned in [70] is a part of the activities of *overlap detection* and *inconsistency detection*. New is the addition of the tracking of inconsistencies and the specification and application of an inconsistency management policy. Tracking is concerned with the *"(a) recording of the reasoning underpinning the detection of an inconsistency, (b) the source, cause and impact of it, (c) the handling actions that were considered in connection with it, and (d) the arguments underpinning the decision to select one of these options and reject the other"*. The specification of a policy includes identifying the relevant stakeholders, identifying the entities responsible for identifying model overlap, and specifying the mechanisms to be used when identifying, and assessing the impact and cost related to handling an inconsistency.

In conclusion, all of the presented *inconsistency management* frameworks identify the need for (1) detecting overlap among models, (2) differentiating between different *types* of inconsistencies, and (3) implementing a mechanism for identifying inconsistencies that allows one to locate the inconsistency, identify to what *class* of inconsistencies it belongs, and store the rationale (or argumentation chain) that led to the decision that an inconsistency is present.

3.3 Approaches to Identifying (Semantic) Overlap

Much research related to the detection of (semantic) overlap (which may be partial) of models has been conducted in a variety of domains. However, the fully automated inference of such model relationships remains an open challenge. Models overlap semantically whenever there exist two or more expressions in models that have a common semantic meaning - that is, given a set of formal languages $\boldsymbol{\mathcal{L}} = \{\mathcal{L}_1, \mathcal{L}_2, ...\}$, a set of corresponding semantic domains $\boldsymbol{\mathcal{D}} = \{\mathcal{D}_1, \mathcal{D}_2, ...\}$ and semantic valuation functions $\mathbf{v} = \{v_1, v_2, ...\}$, the semantically overlapping expressions in each $\mathcal{L} \in \boldsymbol{\mathcal{L}}$ are those elements of the languages that have identical interpretations: i.e., whenever for any i, j: $v_i(\phi_{ix}) = v_j(\phi_{jy})$ is true, where ϕ_{ix} and ϕ_{jy} are well-formed expressions from languages $\mathcal{L}_i$ and $\mathcal{L}_j$ respectively.

Automating the process of identifying semantic overlap is non-trivial, particularly due to the often informal definition of semantics of modeling languages used in practice, and because of the problem of formally representing some semantic domains (refer back to the discussion on this in section 2.1.1.4). Most research in inconsistency management suggests that defining such overlap *manually* or at least *with human assistance* is unavoidable [200]. For instance, in his interference management framework, Finkelstein specifically mentions that overlap identification necessarily requires human intervention [70]. Semantic overlap can only be avoided if the concerns addressed in the various partial models of a system can be separated completely. However, this is well-known to be impossible (there is always *some* relation, no matter how weak, between various aspects of one and the same system), particularly for highly complex systems.

In the related literature, four distinct classes of approaches to overlap identification can be identified: the *exploitation of representation conventions*, *use of a unifying, domain-spanning ontology*, *human inspection*, and *similarity analysis.* Each of these approaches, and notable implementations of these, is presented in the following.

3.3.1 Exploitation of Representation Conventions

The exploitation of (syntactic) representation conventions is the "oldest [sic] and most commonly used form of detecting a model overlap" [200]. Approaches in this class are typically based on *(syntactic) unification algorithms.* Generally, unification algorithms perform a syntactic matching between terms. For instance, when working with distributed databases of logical expressions, predicate matching is often used, where predicate matching is based on symbolic equality of the predicates. This is employed by Finkelstein *et al.* in [71], where visual (or iconic) models (i.e., what is, in the context of this dissertation, generally referred to as a *model* (see section 2.1.2)) are used as partial representations of a system, and are translated to a first-order predicate logic. Overlap among models is then attempted to be identified based on predicate matching. Similar work is conducted by Easterbrook *et al.* in [50]. Overlap identification based purely on name matching is employed for consistency checking of requirements using model checking techniques in [98].

Another, more recent approach is that of utilizing the correspondences defined as part of specifying *model* [36, 37] or *graph transformations* [132, 184]. Typical for defining model transformations is the definition of a syntactic correspondence between elements of a meta-model (see, e.g., [145, 2]). Other related approaches include the definition of *triple graph grammars* (TGGs) [132], which also define syntactic correspondences among (meta-)model elements. Note that the correspondences are (if done on a meta-level) correspondences among *types of things*. However, no possibility (other than naïve inference from type to instance) of identifying that two instances of these meta-model elements represent *identical* things exists (provided a semantic valuation function (see section 2.1.1.4) is not given). The use of TGGs for purposes of explicitly marking overlap among models is presented in, e.g., [83, 79, 80].

Note that the assumptions made by unification algorithms are typically very strong, particularly if they are based on syntactic matching. For example, if matching is done on the basis of names, an overlap cannot be identified if the names are not identical, a spelling mistake is present, or the names being compared are antonyms (in which an obvious relation *should* exist). Also, an overlap is *incorrectly* identified if the names of the elements are intended to be used as homonyms. Given the typical heterogeneity of models and the typical non-existence of an isomorphism between these, even the more recent approaches are limited, particularly when wanting to adopt them for the purpose of identifying inconsistencies. Indeed, in [2] it is mentioned that parallel development of the models would require labor intensive and manual changes to the models.

3.3.2 Use of Unifying, Domain-Spanning Ontologies

Using a common, shared ontology is a second approach to enabling the identification of overlap between models. This approach requires the authors of the models to tag model elements with elements from a common ontology shared among all stakeholders. Model elements are then defined as overlapping if they carry the same tag(s). The approach is used by Boehm and In in [21] for tagging requirement models with elements from the *QARCC* ontology, which defines various quality attributes for software systems.

In Model-Based Systems Engineering research, this has been demonstrated by Hehenberger *et al.* in [97] for the specific case of developing mechatronic systems. The authors develop a set of concepts and relations for the domain of mechatronics, which are used in the definition of consistency rules. A similar approach for the domain of spacecrafts is followed by the *Jet Propulsion Laboratory* (JPL) [183].

A common practical issue related to using common, shared ontologies is that ontologies are models themselves and may, hence, be interpreted differently by the various stakeholders using them (unless, of course, the underlying formalism used for semantics is well-understood by the intended audience and is sound). In addition, modelers are also required to commit to the ontology both in terms of accepting the definitions, and in actively tagging relevant model elements. Particularly the latter can be very labor intensive.

3.3.3 Human Inspection

A third approach to identifying overlap is the use of human inspection. Typically, approaches designed for identifying model overlap using human inspection methods implement a system using which a stakeholder responsible for identifying inconsistencies is aided in some form (e.g., through visual aids, or various ways of representing the models). An example implementation of such an approach is demonstrated in the *Synoptic* system presented in [48]. There, various methods have been investigated including the provision of visual aids for browsing models, graphical selection of elements from two partial models, and the recording of overlap. Other, similar approaches are provided in [123].

Identifying overlap using human inspection can be very exact, but is also highly labor intensive and time consuming [200]. Particularly when developing highly complex systems using (expectedly) highly interrelated models – such as is the case in Model-Based Systems Engineering – such an approach is likely to be too costly to employ. However, there are a number of ongoing research efforts, particularly geared towards visualizing and enabling the analysis of large data sets (such as sets of heterogeneous models) (see, e.g., [12]).

3.3.4 Similarity Analysis

Similarity analysis exploits the fact that modeling languages incorporate constructs which imply or strongly suggest the existence of certain overlap relations [200]. For instance, the fact that two model elements carry the same name does not entail their semantic equivalence, but merely serves as evidence to suggest their equivalence.

In [199], Spanoudakis and Finkelstein analyze structural similarity of models to identify overlap based on the weighted bipartite graph matching problem, where vertices in the two partitions of the graph denote the elements of the models being compared, and the edges of the graph represent possible overlap relations. Weights are then computed based on distance functions, which measure modeling discrepancies in the specifications of the elements connected by the edge with respect to different semantic modeling abstractions [200]. No related work using similarity analysis from the domain of Model-Based Systems Engineering was identified (other than research published by the author of this dissertation: see [107]).

Similarity analysis for semantic overlap identification has also been used in research related to *ontology matching* and *database schema matching.* For instance, Ehrig and Sure use a series of similarity rules (e.g., equal names, equal types, equal sub-concepts) and calculate a weighted similarity score from these [60]. In [17], Berlin and Motro use a combination of methods from probability theory and a scoring technique for identifying similarity among database schemas. In fact, some aspects of machine learning are used in initially finding values for the values of the probabilities involved. Similar work has been published by Doan *et al.* in [45].

Similarity analysis is one of the very few techniques where *abductive* and *inductive inference* (see section 2.2.3.2) is employed rather than relying on deductive reasoning. However, most approaches using similarity analysis use ad-hoc methods for measuring similarity (e.g., weighted sums, arbitrary scores, incorporation of weak assumptions), hence putting their general applicability in MBSE in question.

3.4 Approaches to Inconsistency Identification

In the related literature, a number of approaches to identifying inconsistencies are presented. From the perspective of formal systems and automated proof theory, an inconsistency is present whenever a proposition ϕ and its negation $\neg\phi$ can be derived (see sections 2.1.1 and 2.2). This is in line with the definition given in section 1.2.2.1. However, in practice this is often difficult to apply due to the lack of a formal underlying system, and particularly due to the lack of well-founded and formal semantics. Hence, particularly when considering the agglomeration of a number of (incomplete) models, each representing a different, but related aspect of a system, identifying an inconsistency is no longer as simple as finding a logical contradiction.

Where possible, logical databases are utilized in related research. This is particularly the case for earlier work. Other works mitigate the identified challenge of using purely logical formalisms in a variety of ways. These include rule-based approaches, where the match to a pattern representing the antecedent of an inconsistency rule is sufficient evidence for deducing the presence of an inconsistency. This is sometimes also referred to as an approach where *negative constraints* are applied. Other approaches to inconsistency identification use sets of *consistency constraints* and define an inconsistency to be present whenever such a constraint is violated. These constraints are typically a constraint on the relationship between two entities. Finally, approaches very similar to rule-based approaches, but based on model transformation formalisms are used particularly in more recent work.

3.4.1 Logical Reasoning & Theorem Proving Based Approaches

Approaches based on theorem proving implement the methods presented in section 2.2. The goal of such an approach is to show that a statement ϕ and its negation $\neg\phi$ can be inferred from a single formal system, thereby identifying an inconsistency of the formal system, or to prove that a statement ϕ is not a theorem.

In early work, Finkelstein uses logical reasoning to identify and resolve inconsistencies. In [71], an approach related to the previously introduced VOSE framework (see section 3.2) is introduced. First, views (specifications or models) conforming to viewpoints are translated into, and inter-viewpoint relations are captured as formulas of a first-order logic system. For example, inter-viewpoint relations can be captured as rules of inference similar to those presented in section 2.2.3.1. However, Finkelstein describes this as a (necessarily) largely manual effort [71]. This translation into a FOL enables the representation of potentially heterogeneous model data in a canonical form. Once compiled, a theorem prover is used to infer any logical inconsistencies in the resulting logical database. Since the information in the logical database is an incomplete representation of a system, reasoning is performed under the *Closed World Assumption* (CWA), which assumes that for any proposition ϕ for which the truth value is not known (i.e., here: is not in the database), $\neg\phi$ necessarily holds. If the theorem prover can derive both a statement ϕ and its negation $\neg\phi$ based on the contents of the logical database and the provided inference rules, an inconsistency is said to be present.

Approaches to identifying inconsistencies that are based on theorem proving are the most sound, if the underlying formal system can be defined properly. However, for practical applications (particularly in MBSE) they are difficult to implement. One hurdle is the translation of the models used in MBSE to logical formulas. This would require well-founded semantics and appropriate transformation rules (i.e., mappings). A second hurdle is the assumption that a respective formal system can be formulated in the first place (without making weak assumptions, such as the CWA), which would require a complete set of axioms and inference rules that spans all models and their respective definitions.

3.4.2 Constraint Satisfaction Based Approaches

Some authors view inconsistency identification from the perspective of a constraint satisfaction problem. Common among notable approaches from the related literature is the annotation of models with constraints using a textual constraint language. In [188], the use of a formal language called the *Consistency Constraint Language* (CCL) is described. Conditions are checked based on the current context. This context includes not only the model, but includes an element of time. The constraints can be *variant* or *invariant*, where invariant constraints always apply, and the applicability of variant constraints depends on the specified context. Associated conditions may also be related to a particular view. The language is specific to the AutoFOCUS [116] modeling language and tool-suite, but similar in syntax to the well-known *Object Constraint Language* (OCL) [164], which is used to annotate models that conform to UML.

When considering inconsistency identification as a constraint satisfaction problem, constraints are formulated as (logical) conditions that the model *must meet*. In other words, for a constraint to be satisfied (which marks the absence of an inconsistency) each condition associated with the constraint must evaluate to `true`. Therefore, constraints implicitly (but partially) define states of well-formedness for a model.

Mens *et al.* describe such an approach in [148]. The authors argue that UML 1.5 does not provide adequate support for checking model consistency and model evolution. As part of the research, the authors introduce a profile for UML model consistency and translate the UML meta-model (including the custom profile) into a description logic [7] supported by the knowledge representation, inference and query engine *Loom* [144]. Consistency and evolution rules (which act as constraints enforced at all times and attempt to avoid the introduction of inconsistencies in the first place) are then formulated as production rules.

3.4.3 Procedural, Rule & Pattern Matching Based Approaches

In [142], Liu *et al.* use a set of production rules in the form *"if [condition] then [actions]"* to check for inconsistencies and take appropriate resolution actions. This was demonstrated for a subset of UML. The characteristics of a particular inconsistency are encoded as a pattern in the antecedent (condition) of the rule. The core principle of the approach is the translation of UML models into a representation that can be interpreted by an off-the-shelf rule engine – in this case, the Java-based rule engine *Jess* [73].

Van Der Straeten *et al.* demonstrate the use of patterns to extract instances of inconsistencies from UML models in [222]. In the approach, UML models are first translated to the *Loom* [144] knowledge representation language (which, as mentioned previously, is an implementation of a description logic). *Loom* has a pattern-based query processor, which can be used for querying patterns to return the particular instances of inconsistencies in the models being analyzed for inconsistencies.

Other work, which also suggests the use of patterns to identify inconsistencies, includes the work by Hegedüs *et al.* presented in [96]. A graph-based representation of models in the *Visual Automated Model Transformation* (VIATRA) [35] framework is assumed. Inconsistencies are then defined as graph patterns and interpreted as negative graph constraints. A set of possible graph transformations are associated with each inconsistency pattern for the purpose of resolution (see also [105] for a *design space exploration* perspective on this problem). Egyed follows a similar approach using a framework known as the *Model/Analyzer* [179] in [51, 52, 53].

3.4.4 Approaches Based on Model-Based Rule Definitions

Approaches that consider what may be regarded as *model-based definitions* of inconsistency rules and patterns include the already discussed work by Hegedüs *et al.* [96] and the research presented in [2, 79, 80, 83, 147, 208].

Sunetnanta and Finkelstein devise a framework in which consistency checking rules are defined in a visual manner using conceptual graphs in [208]. Conceptual graphs are a formalism for knowledge representation and are used in representing conceptual schemas and facts. Specifically, Sunetnanta and Finkelstein use conceptual graphs for meta-representations of viewpoints. Both the rules and viewpoint instances are translated into a first-order logic. This process can easily be automated due to the well-known relation between conceptual graphs and FOL formulas. The methods described in section 3.4.1 are then applied to the result.

In [147], Mens *et al.* present an approach to identifying and resolving inconsistencies that is similar to [96] and [105] in the sense that parallel and sequential dependencies between inconsistency rules are analyzed. In his approach, inconsistency rules are specified as graph transformation rules, where the antecedent of the rule is a graph pattern that identifies an inconsistency. Therefore, the antecedent acts as a sufficient condition for the inconsistency.

Giese and Wagner investigate the use of triple graph grammars for the purpose of model synchronization in [83]. TGGs are used for defining the correspondences between meta-models that can be used for the purposes of transformation. A similar approach is followed by Adourian and Vangheluwe in [2], where model transformations are used in preserving the consistency between two models. Gausemeier *et al.* investigate the use of TGGs for consistency management in [79, 80]. The core idea of this approach, which, in its application considers models of mechatronic systems, is the propagation of relevant changes from domain-specific models to a principle solution, and, from there, into domain-specific models. The principle solution is a model similar to what is often referred to as a *system model* in MBSE [72] and acts as a *hub* and only point of integration between models. Therefore, some dependencies between domain-specific models cannot be captured (unless they are a part of the principle solution). As a result, certain inconsistencies between these models cannot be identified. Shah *et al.* counter this disadvantage in similar work by including more domain-specific model data in the system model [191].

3.5 Summary

In this chapter, the results of conducting a review of the related literature on inconsistency management is presented. The specific focus of the investigation is work concerned with identifying semantic overlap in models and identifying inconsistencies. While much related research has been conducted in the domain of software engineering, very little has been published about the applicability of the methods to MBSE.

In the first part of the chapter, frameworks for inconsistency management are introduced. These frameworks stem from the software engineering literature, and all share a number of traits. For instance, common to all of the reviewed frameworks is that the detection of overlap is an inevitable prerequisite to identifying inconsistencies.

Furthermore, the identification of inconsistencies should occur in such a fashion that an inconsistency can be located and classified for the purposes of diagnosing the inconsistency (which includes identifying the root cause of it).

The second part of the chapter introduces a number of state-of-the-art approaches from the related literature on the automated detection of model overlap. The classes of methods analyzed are methods exploiting representation conventions, the use of unifying and domain-spanning ontologies, human inspection, and similarity analysis. Methods that exploit representation conventions are purely syntactic in nature, make strong assumptions and can be error prone. Using unifying and domain-spanning ontologies is identified as a promising method, but requires commitment to the ontology and incurs additional cost due to the necessity of having to hand-label models. Methods relying on human inspection focus on developing visual aids to reduce the complexity and cost incurred with manually defining overlap among models. Approaches using similarity analysis rely on inductive and abductive techniques to identify the most likely areas of model overlap.

Finally, in the third part, methods for inconsistency identification are introduced and reviewed. The investigated methods use logical reasoning and theorem proving (which have been introduced in detail in section 2.2), as well as constraint checking, and procedural and model-based rule definitions. While methods based on theorem proving produce logically correct results, a caveat is constructing and proving the completeness and consistency of an underlying formal system, as well as the translation of the models to the logical formalism. The other approaches introduced produce results that are not guaranteed to be logically correct, and, instead, rely on the definition of sufficient conditions for identifying inconsistencies.

CHAPTER IV

INCONSISTENCIES & INCONSISTENCY MANAGEMENT

In this chapter, inconsistency management as a discipline is briefly introduced and the concept of an *inconsistency* is elaborated upon. As part of the introduction, and to underscore the need for inconsistency management, a rationale is given for why it is valuable to manage *inconsistencies* rather than attempting to preserve *consistency*. It is shown that, while it is desirable to be *as consistent as possible* – both with respect to a formal system and with respect to the environment in which the system will be deployed – it is *impossible* to prove (global) consistency. This argument is followed by a formal definition of what an *inconsistency* is considered to be within the scope of this dissertation. Then a classification of inconsistencies and related types of semantic overlap are given. Finally, a framework for identifying inconsistencies is introduced. Therefore, the primary goal of this chapter is to address research question 1, and to investigate and evaluate hypotheses 1 and 2.

The concepts introduced in this chapter lay the foundations for developing and evaluating an inconsistency identification method. The framework introduced is kept neutral from any specific solution method. By providing a classification of inconsistencies, the completeness of a particular method implementing this framework can be evaluated quantitatively by testing which types of inconsistencies can not be identified (under the assumption that the classification is complete). This framework is inspired by frameworks from the related literature introduced in section 3.2, and is specifically tailored to the case of multiple, heterogeneous models, where an analysis of semantic overlap is inevitable.

4.1 Why Manage Inconsistencies Rather than Consistency?

Within the scope of this research, *inconsistency management* is defined as the *discipline* of *identifying* and *resolving* inconsistencies in models. Inconsistency management provides an alternative view on the problem of *consistency management*, where the goal is to either prove the consistency of a set of models, or to ensure that inconsistencies are never introduced (e.g., by enforcing constraints during construction of a model). In *inconsistency* management, specific instances of types of inconsistencies are sought out and handled.

As introduced in section 2.2.4, a formal system (and the theory that can be formed using it) is complete and consistent if no contradictory propositions can be derived from it and if all true statements of an accompanying formal language can be produced using only the axioms and inference rules of the formal system. As in accordance with previous work [102], inconsistencies with respect to formal systems will be referred to as *internal inconsistencies*. Inconsistencies with respect to the environment in which the system will be deployed (e.g., nature) are referred to as *external inconsistencies*. These two *dimensions* are elaborated upon in the following. Thereafter, as a motivation for inconsistency management, it will be shown that proving internal consistency and the completeness of a formal system is not always possible. It will also be shown that proving external consistency is impossible.

4.1.1 Dimensions of Consistency

In the following, two fundamental dimensions of consistency – namely, *internal* and *external consistency* – are introduced. These dimensions have already been derived and studied in previous work (see [102, 103]), but their definition and impact on the extents to which consistency can be managed in MBSE applications is expanded upon in the following.

The dimension of internal consistency relates to consistency with respect to axiomatic systems that are well understood (e.g., logic systems and mathematics). (Formal) modeling languages are considered a part of such systems. Models that are internally consistent do not violate the axioms and rules of the underlying formal system since they are theorems of the formal system (see section 2.2.3). External consistency imposes an additional constraint: the model of the system must be true to (i.e., be an accurate representation of, and be consistent with) the reality (or world, or environment) that the system will ultimately be deployed in. Differentiating between these two dimensions of consistency is practical, since it allows one to differentiate between consistency issues that occur within the bounds of some well-understood and formally defined system, and those that do not [102].

4.1.1.1 Internal Consistency

In engineering design, models have two main purposes: to specify and to analyze systems. In practice, a large variety of models are utilized. Commonly used in MBSE applications are requirements models, functional models and physical architecture models [72]. Models that are used in specifying a system, and those that are used in its analysis, can be thought of as *abstractions* of a designer's *ideas*, *beliefs* and *preferences* [102, 94]. Ideas are a result of a creative process and allow for alternatives to be specified, whereas beliefs are used in predicting outcomes and are formed based on observations [94, 95, 18]. Preferences are used in evaluating alternatives and ranking them. In MBSE, a key idea is to use only formal models for this purpose. Each model is used to establish one or more *views* on a system that allows a modeler to address specific concerns. These views are established based on a *viewpoint* definition, which also specifies the modeling languages, and hence the formal systems, to be used [121, 72].

A model is *internally consistent* – that is, consistent with respect to a formal

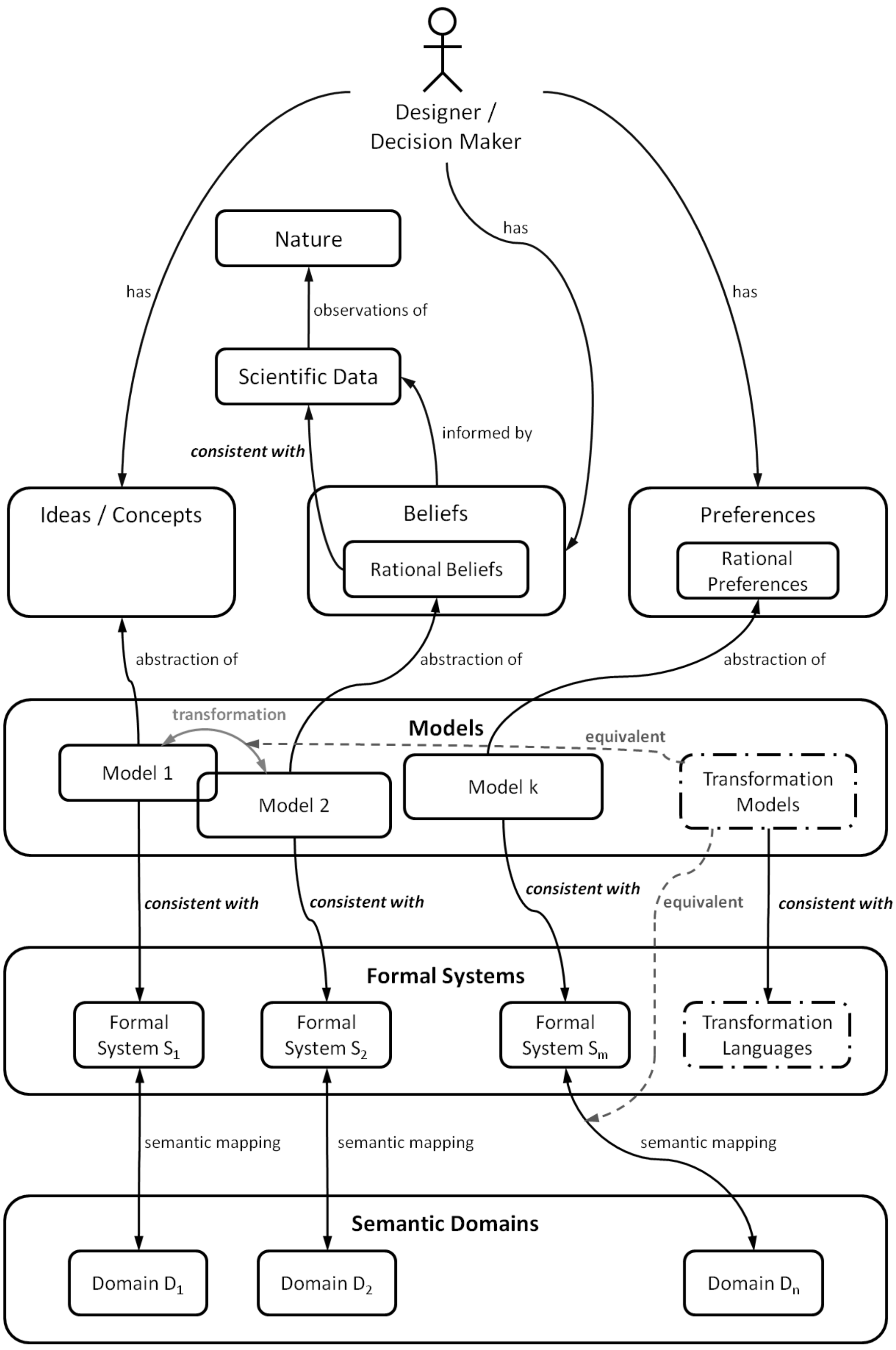


Figure 6: Models as abstractions of the ideas, beliefs and preferences of a designer, and related inconsistencies.

system – if it is well-formed and compatible with *some* feasible world. Feasible worlds are based on the axiomatic systems of logic and mathematics. Such systems are widely accepted to be complete and consistent. Internally consistent models are not in violation with the axioms and rules of the underlying formal system. That is, it is possible to construct a formal proof that demonstrates that the model is a theorem of the system [102].

To exemplify the notion of internal consistency, consider a fair coin toss. A fair coin toss results, by definition, in two equally likely outcomes: heads or tails, each with probability 0.5. Assume that one were to assign a probability 0.6 to the event that the outcome of the experiment is *"heads"*, and a probability 0.5 to the event that the outcome is *"tails"*. This is in contradiction with Kolmogorov's axioms (see section 2.3.1.1), since the sum of the probabilities of all elementary events is greater than 1. Therefore, the model is *internally inconsistent* with respect to the underlying formal system – in this case, stochastics and statistics in mathematics.

The consolidation of all models describing a system – in other words, the *composition* of these models – can be said to form a third, overarching model[1]. This model is internally consistent if both its parts (the models of which it is composed) and the consolidated information and knowledge encoded in the models are consistent. Since it is also a model, a corresponding formal modeling language and formal system must exist.

Consolidating models requires morphisms that translate a formal model from one modeling language into another (e.g., the language of the overarching model). Such morphisms – or *translations*, or *transformations* – are, in model-based development, models themselves which are typically referred to as *transformation models*.

[1]To exemplify the notion of model composition, consider the two expressions (which may be separate models) $S_1 = [a = x + 1]$ and $S_2 = [y = a + 1]$. Composing the two expressions, the following can be derived through substitution: $S = S_1 \otimes S_2 = [y = (x + 1) + 1]$. To be complete and formal, such operations require semantically well-founded languages.

Figure 6 summarizes the relations between a designer's beliefs, preferences and ideas, and models. It also shows important consistency relations.

4.1.1.2 External Consistency

Axiomatizable theories are computable – this was demonstrated in section 2.2.3. Therefore, in theory, the internal consistency of a formal model should be provable by computational means (e.g., using a deductive proof). However, what can be even more challenging is to determine whether or not a model is consistent with what one would observe in the environment in which the system will be deployed. That is: is the model an accurate representation of what one would observe in reality? This can be seen as a separate, but related problem dimension [102]. Consistency issues with respect to a target environment are referred to as *external* consistency issues.

To give an example of an external inconsistency consider, once more, the example of the fair coin toss: an internally consistent model of the experiment of flipping the coin and determining the probability of either events *"heads"* or *"tails"* can be constructed by assigning probability 0.6 to the outcome *"heads"* and 0.4 to *"tails"*. These probabilities may be accurate reflections of the designer's beliefs, and may be consistent with data collected about previous coin tosses with other coins (perhaps, unknowingly, *unfair* coins were used for collecting this data). However, the model is not externally consistent for a fair coin, because the true values for the probabilities are very different (i.e., a very large number of tosses of the fair coin would yield different probabilities).

4.1.2 Formal Issues in Proving Internal Consistency: Consistency in Axiomatic Systems

The definition of theorems as elements of a formal language allows for results in proof theory that study the structure of formal proofs and the structure of provable formulas [113]. The most famous results are Gödel's incompleteness theorems [84]. By representing theorems about basic number theory as expressions in a formal language, and then representing this language within number theory itself, Gödel was able to construct examples of statements that are neither provable, nor disprovable from axiomatizations of number theory. Gödel also showed that any consistent axiomatizable theory which can encode (finite) sequences of numbers, the consistency is not provable in the system. Both of these results, and important implications for (in)consistency management, are discussed in the following.

4.1.2.1 Gödel's Incompleteness Theorems

Typically, it is taken for granted that formal proofs (see section 2.2.3) always determine with perfect precision whether or not some expression (e.g., a model) belongs to a particular language – i.e., whether it is consistent with the formal system. However, it can be shown that this is *not* the case for all formal systems. In his famous work, *Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme* [84], Gödel has proven that not all formal systems are able to produce all true statements about themselves. Gödel showed this with his *first incompleteness theorem*, which states that *"any adequate axiomatizable theory is incomplete"*. The proof can be constructed as follows: let $PROV$ be the set of numbers which encode sentences which are provable from a given set of axioms from a formal system. Thus, for any sentence s, s is in $PROV$ *iff* s is provable. Now consider the sentence s = "This sentence is not provable". Being a truth bearer, this sentence is either `true` or `false`. s is in $\neg PROV$, since it is, by its semantics, not provable. However, if s is `false`, then s is provable. However, this is a contradiction, since provable sentences are always `true` [113].

An important conclusion, highly relevant to the management of consistency of formal models, can be drawn from Gödel's first incompleteness theorem: *"for some (modeling) languages, there are utterances (models) that are true, but their truth is not* *provable"* [84, 113]. In other words, there are some models (or, generally, expressions) that are true and consistent with an underlying formal system, but any attempt at proving their consistency leads to a contradiction (i.e., an inconsistency).

Gödel's second incompleteness theorem states that *"in any consistent axiomatizable theory which can encode sequences of numbers (and, thus, also the syntactic notions of "formula", "sentence" and "proof") the consistency of the system is not provable in the system"* [84, 113]. This theorem is a result of discovering that the sentence "This sentence is not provable" is provably equivalent to the formal statement that a system is consistent (see [84] for the full proof).

The second incompleteness theorem has several interesting implications for formal theories. In particular, a theory $\mathcal{T}_1$ cannot prove the consistency of any other theory $\mathcal{T}_2$ that proves the consistency of $\mathcal{T}_1$. This is because such a theory $\mathcal{T}_1$ can prove that if $\mathcal{T}_2$ proves the consistency of $\mathcal{T}_1$, then $\mathcal{T}_1$ is in fact consistent. If $\mathcal{T}_1$ were in fact inconsistent, then $\mathcal{T}_2$ could prove that there exists a contradiction in $\mathcal{T}_1$. But, if $\mathcal{T}_2$ also proved that $\mathcal{T}_1$ is consistent, $\mathcal{T}_2$ itself would be inconsistent. This reasoning can be formalized in $\mathcal{T}_1$ to show that if $\mathcal{T}_2$ is consistent, then $\mathcal{T}_1$ is consistent. Since, by the second incompleteness theorem, $\mathcal{T}_1$ does not prove its consistency, it cannot prove the consistency of $\mathcal{T}_2$ either. This means that it is impossible to prove, for example, the consistency of *Peano Arithmetic* (PA) [113] using any finitistic means that can be formalized in a theory of which the consistency is provable in PA. For example, the theory of *Primitive Recursive Arithmetic* (PRA), which is widely accepted as an accurate formalization of finitistic mathematics, is provably consistent in PA. Thus PRA cannot prove the consistency of PA [113].

The interest in proving consistency lies in the possibility of proving the consistency of a theory $\mathcal{T}_i$ in some other theory $\mathcal{T}_j$, which, in some sense, is less doubtful, or weaker, than $\mathcal{T}_i$. However, if $\mathcal{T}_j$ were in fact *inconsistent*, $\mathcal{T}_i$ can be proven to be consistent, since inconsistent theories prove everything [138, 69], including their own consistency.

Proving the consistency of $\mathcal{T}_j$ requires some other consistent theory $\mathcal{T}_k$, and so on.

For managing the consistency of a formal model, this result has the important implication that any attempt at proving consistency (which requires a consistent theory) is, at least formally, in vain. An underlying formal system cannot be proven to be consistent and is, due to its axiomatic nature, provably incomplete. Therefore, proving the consistency of a formal model is impossible[2].

4.1.2.2 Composing Formal Models

In MBSE, the complexity associated with designing a technical system is typically managed, in part, by some decomposition mechanism. In practice, this leads to the various stakeholders involved in the design and development process to establish views on the system from different perspectives by creating models to address their specific concerns of interest. This leads to an often large number of partially overlapping and different models which, at least in practice, are typically based on different modeling languages.

Since it is impossible to completely separate concerns and decouple models, models will always overlap or be dependent on one another in some form [106, 174]. Therefore, (in)consistency proofs should not be limited to individual models, but the *agglomeration* of all models should be analyzed. Hence, an important part of managing the consistency of a set of models is forming their composition. Composition and related consistency issues have been briefly investigated from a theoretical standpoint in [106, 104] and their importance for model integration is further outlined in [24]. In the following, a brief overview is given to introduce relevant terminology.

Assuming there exist n models m_i that are to be composed, then a model m_{sys}

[2]Note that this does not rule out consistency proofs altogether, nor does it diminish the importance of proof theory. The only consistency proofs that should be ruled out are those that can be formalized in the theory which is proved consistent.

encompassing all aspects captured about a system can be defined as:

$$\mathrm{m}_{sys} = \mathrm{m}_1 \otimes \mathrm{m}_2 \otimes ... \otimes \mathrm{m}_n \ .$$

Here, the operator $\otimes$ is used to denote the *composition* operation for models of various types. A similar composition operator can be defined for the composition of the accompanying languages (say there are m formal languages under consideration):

$$\mathcal{L}_{sys} = \mathcal{L}_1 \otimes \mathcal{L}_2 \otimes ... \otimes \mathcal{L}_m \ .$$

Implementing composition operators in a formal fashion requires that the formal languages being composed are semantically well-founded [24, 104]. Without this quality, simpler statements cannot be folded into more complex ones. Such operations require an *understanding* of the models and their parts. For instance, there may be expressions in two distinct languages that are syntactically different, but have the same meaning. As an example, consider the syntactically different, but semantically equivalent statements about a *date* from section 2.1.1.4. Composing n semantically related expressions leads to the formation of a $(n+1)$th expression, possibly syntactically different, which carries the combined meaning of the n expressions. Composition must be defined in terms of the target language $\mathcal{L}_{sys}$, which carries its own syntactic representation. Formally, a composition can be viewed as a morphism $h_c : \boldsymbol{\mathcal{L}_s} \rightarrow \mathcal{L}_t$ from a set of source languages $\boldsymbol{\mathcal{L}_s} = \{\mathcal{L}_1, \mathcal{L}_2, ...\}$ to a target language $\mathcal{L}_t$. Therefore, one can also think of a composition (at least in a directional sense) as a translation operation.

In practice, the semantics of such compositions are often defined in an ad hoc fashion due to the lack of formality in the definition of modeling language semantics. For instance, most *tool chains* implement such mechanisms using procedural code, supporting only a limited set of modeling languages, and making weak assumptions about the structure of the corresponding models. Composition-related consistency issues are also, to some degree, a motivating factor for the more formal *model* and *graph transformations*. These can be seen as the implementation of the corresponding morphisms (or mappings), either for the purpose of *model synchronization* (e.g., using triple graph grammars (TGG) [132, 79, 83]) or *translation* between formal systems (see, e.g., [168, 202, 191, 109]), often for the purpose of model integration. However, a limitation is that these transformations are typically only defined for pairs of modeling languages and, in most cases, assume the existence of a structural isomorphism. In addition, strong assumptions about the structure of the target model are made (such as shown as a limitation in [2]), which is, in part, a result of a lack of formality of the definition of the semantics of the modeling languages. Therefore, in general, while valuable for certain cases, they are not *true* implementations of the underlying composition operators, but mere ad hoc translations based on structural properties.

In summary, a prerequisite for proving the internal consistency of a set of *heterogeneous* models – that is, models that are utterances of different formal modeling languages with incompatible meta-models – is the definition of an additional language (and formal system) using which the result of composing the models can be represented. In addition, the theory constructed using this additional formal system would need to be consistent. However, as discussed in the previous section, this is impossible to prove in all cases. From a practical perspective, defining such a language is non-trivial; in the related literature, research has been done towards identifying such a formalism [87, 210, 181]. However, all of these have, to the date of writing this dissertation, limitations with respect to their expressiveness and interpretative qualities (i.e., the definition of their formal semantics). Furthermore, the expressiveness of these languages is generally limited in that only a limited subset of composeable modeling languages is supported.

4.1.3 Formal Issues in Proving External Consistency

Models are, by definition, abstractions [82]. Being an abstraction, a number of assumptions also flow into models. Therefore, the challenging question is: does a model accurately reflect and appropriately abstract a process that takes place in a particular reality? In other words: is a model *externally* consistent?

For technical systems, this question cannot be answered with certainty. A designer abstracts his or her beliefs using models. These beliefs are informed by observations of nature (i.e., scientific data). Models can be derived from scientific data using a variety of methods: for example, *regression* is a popular technique in engineering [30]. However, while models may *"fit"* certain observations, they may not be accurate representations of reality due to erroneous initial assumptions or structural errors. For instance, Johannes Kepler observed the motion of the planets and derived models for planetary motion by abstracting the scientific data he collected during many long nights [64]. Therefore, he has abstracted a process that takes place in nature. Others before him have tried to do the same and constructed very different models. However, his models predict the actual behavior more accurately[3]. However, they do not describe the actual process with perfect precision [64, 228].

Although based on logical operations, similar observations can be made about models of software systems. Software is deployed on platforms that perform computations based on binary logic. Theoretically, valid computer programs are therefore, in some sense, deterministic in their behavior. However, when considering the hardware that a software code is designed to run on, stochastic processes are revealed. Physically, state changes between `1`'s and `0`'s happen based on a stochastic process and are, hence, non-deterministic. Accounting for the randomness involved in the

[3]One of the reasons for this is that he did not use a geocentric perspective (such as was the case in the Ptolemaic system of astronomy, for instance). In other words, he did not assume that the stars and planets move around Earth, but that celestial bodies move around the Sun [64].

accompanying events requires an abstraction of this process – i.e., a model – to be created. In the (highly unlikely, but theoretically still possible) event that a bit is interpreted wrong, the software produces non-deterministic results.

In practice, a small deviation from observed behavior is acceptable in most cases. Such models are considered *"sufficiently accurate"* and *valuable* models of real processes. However, such models are typically only valid under very specific conditions and hold only if a set of relevant assumptions hold true. These conditions and assumptions show the dependence of a model on time, place and other boundary conditions[4]. Assumptions are necessary, since humans lack perfect knowledge of nature and the processes therein. However, the downside of this is that models cannot be *proven* consistent with respect to the respective (not *fully* understood) environment [113, 102]. Therefore, formally proving the consistency of a model, and proving its validity with respect to the target environment, is impossible. The best one can do is to attempt to identify mismatches between reality and the model – that is, identify certain *types of possible (and relevant) external inconsistencies.*

4.2 Characterizing & Classifying Inconsistencies in Formal Models

In the previous section, two distinct dimensions of consistency are introduced: internal and external consistency. Internal consistency refers to the state of consistency of an axiomatizable formal system, while external consistency refers to actual processes in some reality or target environment (e.g., nature). The conclusion of the previous section is that (complete, global) consistency cannot be claimed. However, what *is* possible is to prove *inconsistency.* This motivates the need for, and demonstrates the value of, inconsistency management.

[4]A classical example from aero- and fluid-dynamics for this is the use of different models for describing flow of a medium at different Mach numbers: different models are valid under different flow conditions.

In the following, inconsistencies are characterized and classified. Based on this characterization and classification, a definition for the term *"inconsistency"* is given. As part of the characterization, features and other observable properties are extracted from a number of example inconsistencies, which are studied in depth. A classification of inconsistencies is then presented, which is based not only on the characterization, but also the critically evaluated inconsistency classifications from the related literature.

4.2.1 Examples of Inconsistencies

In section 1.2.2.1, inconsistencies are defined as the quality of *"having parts that disagree with each other"* and as *"assertions about aspects of the system [...] which are not jointly satisfiable"*. Given that these definitions are quite broad, an exhaustive list of examples of inconsistencies in models is impossible to present within the scope of this dissertation. Instead, several representative examples that are believed to cover a broad spectrum of types of inconsistencies will be introduced: a language *well-formedness violation*, the violation of a *guideline*, and a situation where *facts (or assertions) that cannot jointly be true* exist.

4.2.1.1 Example 1: Well-Formedness Violation

The first example describes a well-formedness violation and is adopted from [56]. In the example scenario, a UML class diagram and UML sequence diagram have been constructed. In the example depicted in figure 7, the cardinality of the relation between the classifiers does not match the specific scenario described in the sequence diagram. That is, in the sequence diagram, a `destroy` call would lead to a cardinality `0` for the particular instance which, by definition of the underlying model, is not well-formed. This can be understood as a type of violation of a *structural* or *syntactical constraint* defined by a corresponding modeling language.

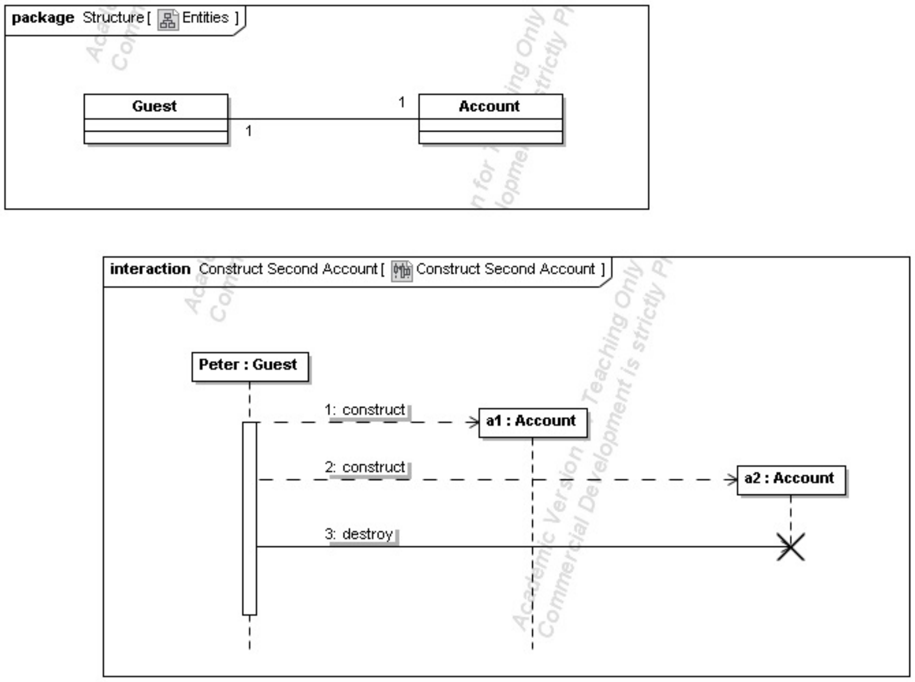

package Structure [Entities]
Guest
Account
1
1
interaction Construct Second Account [Construct Second Account]
Peter : Guest
1: construct
a1 : Account
2: construct
a2 : Account
3: destroy
Academic Version for Teaching Only
Commercial Development is strictly Prohibited

Figure 7: Example inconsistency with respect to well-formedness: cardinality of generic classifiers does not match specific scenario (adapted from [56]).

This particular example is a representative scenario of a situation where a composition of the two diagrams (or models[5]) would lead to an expression that is not well-formed. One can think of this as a scenario where there are two statements that cannot jointly be true by comparing the statements *"every guest must have exactly one account"* and *"Peter (a guest) has no account"*. While such an inconsistency may, by some, be considered to be a more prominent example of limited tool-support for a modeling language, it is, nonetheless, realistic in practical scenarios. Inconsistencies such as these are common due to the often (semi-)formal nature of some modeling languages. This is particularly the case for general purpose, and extensible modeling languages such as the UML [166], where semantic variation points are allowed. For instance, the concept of stereotyping introduces the possibility of adding additional syntactic constructs. However, UML has no formal means of defining the semantics

[5]Note that in the software engineering community, a great number of authors consider diagrams, such as UML diagrams and UML sequence diagrams, to be separate models.

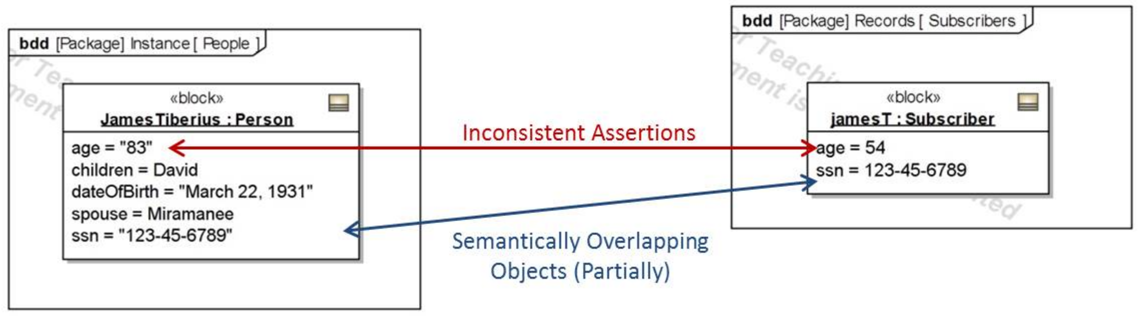


Figure 8: Example scenario where the consolidated set of information available about a distinct entity in two separate models is inconsistent, here modeled as SysML instances.

of these relationships, which leads to the introduction of ambiguous concepts.

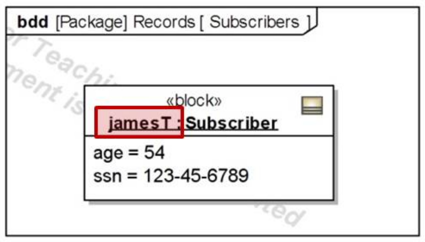
4.2.1.2 Example 2: Conflicting Assertions

A second example of an inconsistency is that of a pair of conflicting assertions in two separate models about a semantically identical entity. Representative of such an inconsistency is the situation where two statements are made that constrain a semantically equivalent property. Consider the example from figure 8: there, two sets of personal information are given (for purposes of simplicity, both are represented as SysML [160] instances). The information represented in these two instances overlaps, because assertions involving semantically equivalent (functional) properties are made. These sets are in conflict because both specify an age (a functional property (every person has only one age)) for (semantically) the same person, but the ages do not match.

Detecting such inconsistencies is non-trivial, since it also requires knowledge about meta-level constraints: firstly, it must be known that any person can only have one age (i.e., one property that, semantically, refers to the age of a person). If this constraint were not explicitly known, the inconsistency cannot be detected since the available information and knowledge does not allow one to conclude that a person with two ages cannot exist. However, if knowledge about such constraints is given, then an inconsistency (or conflict) can, at least in theory, be derived by comparing the ages. The consolidated set of assertions is inconsistent if both constraints on the property *"age"* are not *equivalent*. However, a prerequisite to this is the identification that the property *"age"* refers to the semantically identical property in this case; that is, not only is it the same *type* of property (a property denoting the age of a person), but it is also the same *instance* of it. That is, the context is the same. For the example in figure 8, this is non-trivial to detect for a variety of reasons: while, as a human, one may infer that the person is indeed the same, a human might conclude this based on analyzing the semantic context and implicitly known relations. To a computer, this is more challenging: analyzing the context computationally, one would reveal that the type of the predicated entities (`Subscriber` and `Person`) are not equivalent (however, to a human, these would, in all likelihood, be related in some fashion). A computer would need to understand that the concept of a *person* implies an age and, at least in the United States, a *social security number* (ssn) *uniquely identifies* each person. The computer would also need to understand that a *subscriber* refers to a person who is committed to something.

4.2.1.3 Example 3: Violation of a Standard Practice

Another example of what, within the scope of this dissertation, is considered an inconsistency is the violation of a standard practice, best practice, convention or guideline. Guidelines and best practices are important for a variety of reasons: for instance, for capturing expert knowledge (e.g., *Design for Manufacturing* (DFM) rules [209]) and the readability and maintainability of development artifacts (e.g., coding or style guidelines, and part numbers). The example studied is that of the violation of a *naming convention* which may be classified under violations of *style guides* or *best practices*. Naming conventions are used throughout practice in development processes, both in software engineering and in large complex system development. Primarily,



Rule (informal):

1. Names of SysML blocks and their instances must start with an upper case letter.
2. Each word contained in a compound name must start with an upper case letter.
3. The name may not start with a numeric symbol.
4. Only symbols from the English alphabet (A-Z, a-z) and numbers 0-9 may be used.

Figure 9: Illustration of an example inconsistency with respect to a specified naming convention for SysML blocks.

these are used for readability and to ensure compatibility with legacy systems.

In software and systems engineering, naming conventions are often used for purposes of readability: for instance, a common standard is to use the `UpperCamelCase` convention for names of SysML blocks and UML classes, and the `camelCase` convention for names of class attributes (properties) and function names. In figure 9 an example is given where such a naming convention is explicitly stated and expected to be followed, but has not been followed. This represents an inconsistency because, similar to the examples given beforehand, the rule and the model are *"not in agreement with one another"* (see definitions in section 1.2.2.1), and allowing for both to be true at the same time leads to a contradiction.

In the domain of mechanical engineering a similar example can be found: part numbers typically follow a certain convention so that parts can more easily be identified (i.e., uniquely). However, in practice, part numbering standards are frequently not followed, leading to ambiguity in part number assignments and potential rework[6]. Note that in practice naming conventions (and other standard practices and guidelines) are wide spread and very common in their use. However, very few methods exist for enforcing, or checking the conformance thereto. In most instances, naming conventions are stored in textual form, the convention primarily explained by example,

[6]That this is a realistic (and, in terms of automation, unsolved) problem in practice has been discovered by the author during numerous conversations with experts from industry at countless summits, workshops and conferences.

and designed only for a human to understand.

4.2.2 Characterization & Definition

In section 1.2.2, the notion of *inconsistency* and what it means to be *inconsistent* is first introduced. There, the definitions given are left, intentionally, quite open, broad and verbose. However, now that sufficient background has been introduced, a more formal definition can be given. To do so, features and properties of the inconsistencies presented in the previous section are first extracted. Thereafter, the notion of what is defined as an inconsistency within the scope of this dissertation is formalized.

4.2.2.1 Features & Properties

In the following, features indicative of an inconsistency are extracted from the provided examples. Thereafter, general features of inconsistencies are derived.

Example 1 (see figure 7) is illustrative of an inconsistency with respect to well-formedness of a model. In the example, the constraints defined by a model of two related entities (which are imposed on a model of a specific scenario in which there exist specific *instances* of these entities) are violated. One prominent feature of this example is that the models overlap: *"Peter"* and *"Guest"* are related to one another, as are *"a1"* and *"a2"* to *"Account"*. The overlap is signified by a number of *instance of* relations between the diagrams – more specifically, *"Peter"* is an entity in the class of *"Guest"*s, and *"a1"* and *"a2"* are entities in the class of *"Account"*s. This requires an interpretation of the diagrams. Additionally, more complex relations exist. Strongly related to the illustrated inconsistency is the relation between the number of instances of *"Account"* that *are* associated with *"Peter"* in the given scenario, and the number of such relations that are *allowed.* Determining the former requires an interpretation of the class diagram, and for the latter, an interpretation of the sequence diagram. Identifying the inconsistency requires an interpretation of the *relation* between the two values: in order for them to be *not* inconsistent, they must be equal. The prominent feature in the example is that *there exists a state in which an assertion is made about an individual belonging to a class, which is not compatible with an assertion made about all individuals of the class.*

The second example (see figure 8) illustrates a similar scenario, with the difference being that an inconsistency exists at the same *level*. In this case, *similar* assertions are made about *semantically equivalent* entities (or, philosophically speaking, objects [135]), and the assertions cannot jointly be true. Note the use of the word *similar*: in both cases, assertions are made about a property referred to as *"age"*. Semantically, this is the age of a specific *person* (i.e., an individual in a class of people). The statement that a person's age is 83, and the statement that a person's age is 54 are *semantically different* – i.e., not equivalent – and suggest that a *different* person is referred to (since any person can only be associated with one age) (and unless 54 is equivalent to 83 which, according to a standard mathematical interpretation is not satisfiable). Standing alone, these statements are not inconsistent. However, given the additional information that, in both cases, the age of one and the same person is being asserted, and that a member of the class of *Person*s can only have one age, the statements are inconsistent. This additional information asserts that the syntactic properties *"age"* are semantically equivalent and, by inference, the value assigned to these must be semantically equivalent as well, unless an interpretation of these values leads to the conclusion that they are not. Note that a particularly interesting feature of this example is that the *class* of things being referred to when considering the fact that a person may only have one age, is neither the `Person`, nor the `Subscriber` class. Clearly, both classes are different syntactic types. However, both diagrams contain statements that are, when *interpreted*, related to the *semantic concept* of a person and his or her age.

Example 3 (see figure 9) is an example of an inconsistency due to the non-conformance to a standard. This situation can analogously be interpreted as a situation in which assertions are made about a general *class of things* – here, expressed as a constraint on the names of SysML blocks – and an individual that is *intended to be an individual in the class*. Note the use of the word *intended*: semantically speaking, both statements are *different*. However, when considered together, and assuming that the constraint is quantified over *all* SysML blocks considered, the particular SysML block is *intended* to be an individual of the class of *well-formed* SysML blocks (well-formed with respect to all imposed constraints), but is not a member of it since it violates at least one constraint that is imposed over it. Again, a situation arises in which assertions about related things are made that cannot jointly be true. Particularly interesting about this example is that it is similar to the situation in which a requirement is violated – generally, requirements constrain a specification. Here, the naming convention may be interpreted as such a requirement on the structure of SysML models.

Overall, all three examples demonstrate situations in which *statements are made that cannot jointly be true*. The compared statements can be about *classes of things* and *individuals in a class*, where the *class* can refer to both the syntactic and semantic notion of a class. This is indicative of the existence of *semantic relations*. Additionally, in all three examples, the specific parts of the models involved in the inconsistency require some level of *interpretation*. Furthermore, a general property of these inconsistencies is that certain *assumptions* need to be made when formulating and argument about the presence of a particular inconsistency, and that only the information and knowledge that is explicitly modeled, and that can be interpreted, can be used for identifying an inconsistency. Finally, the examples from section 4.2.1 can be abstracted and generalized to form *types* or *classes* of inconsistencies.

4.2.2.2 Defining "Inconsistency"

In the previous section, features and properties were extracted from examples of inconsistencies. This section abstracts these features and proposes a definition for the term *inconsistency* that is meant to be valid within the context of this dissertation:

Definition 4.1. *An inconsistency is a state of conflict in which, under an interpretation, two or more related statements are accepted as true, that cannot jointly be true. Given the information and knowledge available for reasoning, and a consistent interpretation, sufficient evidence exists to conclude that no situation exists in which the particular set of statements under consideration can be true.*

The first part of the definition is directly derived from the common observation of the example inconsistencies. Similar to the classical definition of an inconsistency in a formal deductive apparatus, an inconsistency is said to exist whenever two expressions are derivable that, given an interpretation, cannot jointly be true. The second part of the definition refers to the fact that assumptions are necessary when concluding that an inconsistency exists. This is the case when the definition of the underlying formal system is incomplete, the (model theoretic) interpretation is defined only partially (particularly that of the agglomeration of models), and to acknowledge the fact that axiomatizable theories are (generally) incomplete (see section 4.1.2.1). Note that the phrase *"consistent interpretation"* is used to indicate that different formal models generally have different interpretations, but the interpretation of the product of composing the models must be consistent in itself. This results in a recursive definition of inconsistency, since (as explained in detail in section 2.1.1) formal semantics (semantic domain and mapping) are defined by a corresponding formal system themselves. The second part of the definition is also practical for an abductive view on inconsistency identification, such as is the case on this research. Definition 4.1 may be viewed as a generalization of the definitions for inconsistency from the related literature given in section 1.2.2.1.

4.2.3 Classification of Inconsistencies

Various examples of inconsistencies have been presented in section 4.2.1. However, as mentioned previously, each of these may be viewed as a specific instance from a distinct *class* of inconsistencies. This observation already supports hypothesis 2. In this section, a classification of inconsistencies is presented that is based on the insights of the previous sections.

4.2.3.1 Classifications from the Related Literature

In the related literature, numerous inconsistency classifications and dimensions of (in)consistency are proposed [62, 220, 148, 188]. Common to these is a dimension which differentiates *syntactic* and *semantic* inconsistencies.

In [148], Mens *et al.* propose a classification of UML model inconsistencies, in which three distinct dimensions are differentiated: the first dimension distinguishes between horizontal, evolution and vertical consistency, the second dimension between syntactic and semantic consistency and the third dimension between observation and invocation consistency. A distinction is also made between various kinds of structural and behavioral inconsistencies. Note that, in the presented work, horizontal and vertical refer to levels of abstraction in UML models. Egyed [52] uses *rules* to identify and resolve inconsistencies (see section 3.4.3) and differentiates between *domain-* and *application-specific* rules, where some relate to instance level objects, while others relate to meta objects. In his doctoral dissertation, Egyed derives a more elaborate classification of inconsistencies – however, this classification is specific to the considered case of UML model inconsistencies [56]. A more general classification of inconsistencies, independent of any modeling language, can be found in [200]: the types of inconsistencies considered are inconsistencies with respect to *well-formedness*, *description identity* (of which figure 8 is a concrete example), *application* *domain*, *development compatibility* (e.g., existence of unified data types), and *development process compliance.*

In summary, most proposed classifications are the result of research in software engineering and, in most cases, strongly related to UML models, which introduces a bias towards specific aspects and features of UML. Therefore, a different classification is proposed in the following, where the aim is to formulate inconsistency types that are independent of specific modeling languages.

4.2.3.2 A Classification for Inconsistencies in Model-Based Systems Engineering

In the presented examples, the class of *well-formedness violations*, *contradicting statements* and *non-conformance to a style guideline* were identified. Well-formedness violations can be both syntactic and semantic in nature. Contradicting statements are a specialization of logical contradictions, in that two propositions are compared. Related to this are the class of *inconsistent predictions*: given identical prior beliefs and observations, as well as identical causal assumptions, predictions are inconsistent if they yield a different result.

All of these types of inconsistencies have a profound effect on the *value* of both a set of models and the artifact intended to be described, since it is impossible for the described *system* to exist in any logical (or rational) world. Since this entails that the system cannot possibly exist in any feasible world, it must, by inference, be inconsistent with perceived reality. In contrast to this, a violation of a best practice, guideline or style guide has an impact on the value of the models and the system, but it is not necessarily impossible for the system to be deployed in a target reality. In light of this, a distinction is made between *strong inconsistencies* and *weak inconsistencies.*

Figure 10 illustrates the classification for inconsistencies developed as part of this research. The classification is depicted by a class diagram, where arrows denote generalization relationships. Note that, in addition to the classes introduced so far, a

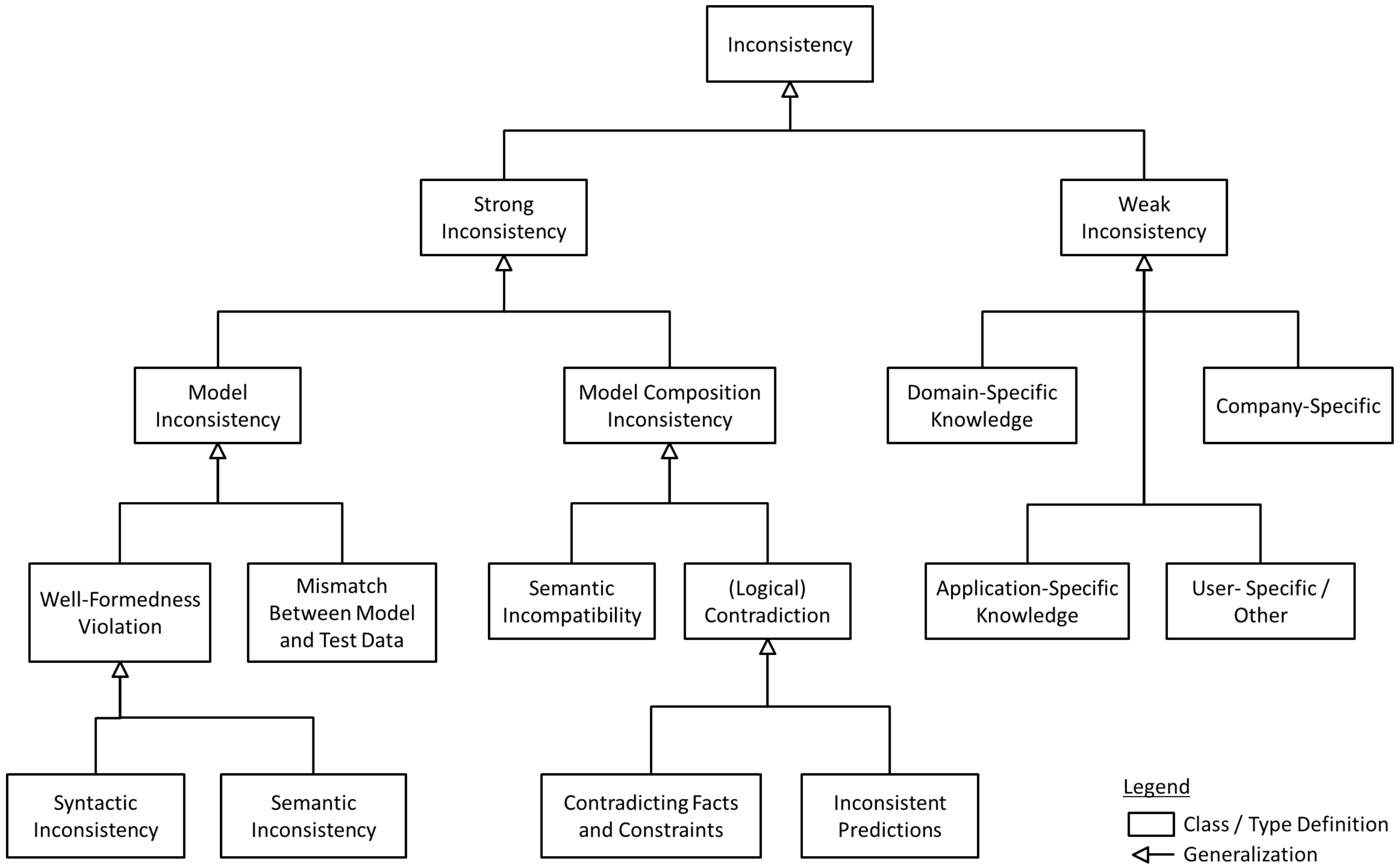


Figure 10: Classification of inconsistency types illustrated as a type definition hierarchy using a simplified class diagram syntax.

number of additional types of inconsistencies may be identified: two of these are *mismatches between model and test data* and *semantic incompatibility*. A mismatch between model and test data can be seen as a class of *external inconsistencies*, which prove that a model is inconsistent with respect to a set of observations. Semantic incompatibility refers to the issue of incompatible interpretations. That is, it is impossible to construct an interpretation that is free of inconsistencies when forming the composition of two models. Contrary to some classifications from the related literature, model *evolution* inconsistencies are not taken into account. The rationale for this is that only the state of the most up-to-date information should be considered when checking for inconsistencies. Process-related inconsistencies were also not made explicit, since these are considered inconsistencies with respect to a particular model of the process.

Instead of defining classes related to *style guide violations* and *violations of best practices*, these classes of inconsistencies are abstracted further and the classes of *domain-*, *application-*, *company-* and *user-specific* inconsistencies are suggested.

Note that, in general, the conclusion of this research is that a closed set of inconsistencies cannot be identified. The category of *strong inconsistencies* can be interpreted as forming a closed subset – however, this is only valid under the assumption that all modeling languages under consideration have a formally defined syntax and semantics, and that there exists a universally valid composition operator. A set of inconsistencies that is open (by definition) is the set of *weak inconsistencies*.

An interesting quality of the classification is its recursive nature and interplay between kinds of inconsistencies. To illustrate this, refer back to the example inconsistency from figure 9: the constraint imposed over names of inconsistencies is a *model* for satisfiable names of SysML blocks. However, being a model, it is subject to the types of inconsistencies identified in the classification. Secondly, consider the interplay between *Model Inconsistency* and *Model Composition Inconsistency*: the result of a composition is a model, which is subject to *Model Inconsistencies*. However, the morphism used is subject to *Model Composition Inconsistencies*. With the definition of models and model compositions being models themselves, they are subject to the same types of inconsistencies as illustrated in the figure.

4.3 Inconsistency Identification: a Framework

Based on the identified limitations of consistency proofs, and the characterization and classification of inconsistencies, a framework for identifying inconsistencies is presented in the following. The framework takes into account the tasks that were identified as essential to any method for inconsistency identification in MBSE.

4.3.1 Practical Limitations of Deductive Proofs in MBSE

As outlined in section 2.2, proving the consistency of an expression (say, a formal model) with a formal system entails a process of (logically) demonstrating that the expression is a theorem of the formal system. That is, the expression must be well-formed, and, if accepted as true, it must be true under the given interpretation. The syntactical proof of well-formedness can be done in one of two ways: either by starting from the axioms, successively applying inference rules to produce expressions until the target expression is reached (forward chaining), or by working backwards from the inference rules to axioms (backward chaining). However, while this leads to logically correct conclusions about the syntactic well-formedness of the expression, its practical application is non-trivial: firstly, the underlying formal system must be complete and consistent (see section 2.2.4). Secondly, there must exist some algorithm that can determine, in finite time, whether or not the given expression is well-formed. The latter is strongly related to decidability and *terminability*.

Recalling the discussion in section 4.1, one of the primary motivations for *inconsistency management* is the fact that the consistency and completeness of a formal system cannot be proven in general. Furthermore, recalling the properties of formal systems discussed in section 2.2.4, not all languages are *decidable* – that is, it is not possible to construct a single algorithm for all languages that results in the (logically correct) conclusion that the expression is a theorem or non-theorem. In fact, in many cases, formal systems are only semi-decidable or undecidable, leading to the problem that an attempt to prove the (in)consistency of a non-theorem (i.e., an expression that is *inconsistent* with a given formal system) may result in the corresponding algorithm to never terminate (recall from section 2.1.1 that all but a simple subset of languages are infinite, hence leading to an infinite number of possible expressions that can be formed).

Logically correct proofs, such as those outlined above, require the use of a *deductive* apparatus, which necessitates that the mentioned properties of formal systems must be satisfied. Even if satisfied, the problem of decidability still severely limits the applicability of the use of theorem proving for inconsistency identification. These limitations apply especially to the case of MBSE, where the management of inconsistencies in the agglomeration (result of composition) of a set of heterogeneous models is of interest. A complete and consistent underlying formal system for the result of the composition of a set of models is non-trivial to define[7], partly due to the disparity and multi-disciplinary nature of engineering systems, and hence the heterogeneity of the models used.

4.3.2 Identifying Probable Inconsistencies using Abductive Reasoning

In this research, an *abductive* rather than a *deductive* perspective on *identifying inconsistencies* is taken. That is, rather than attempting to logically deduce the (in)consistency of an expression with a formal system, an argument is formulated to reason about the *possible inconsistency* of an expression with a formal system. Recalling the introduction to abductive reasoning in section 2.2.3.2, abductive reasoning

[7]Recall the discussion in section 4.1.2.2, where difficulties of, and attempts at formulating a *universal formal system* are briefly introduced.

is an *inexact* reasoning method that is explanatory in nature. That is, the conclusion is accepted to be the best *explanation* for the observations made. This is different from deductive inference, where a conclusion is *logically entailed* from an underlying set of axioms and inference rules. Note that this also means that conclusions drawn from abductive reasoning are not guaranteed to be logically correct.

Within the presented framework, the space over which observations for abductive arguments can be made is the space of all formal models under consideration. That is, an observation manifests as a particular part of one or more models. Therefore, an abductive argument about an inconsistency in a model starts with a number of observations about models and, based on the observations made, leads to the conclusion that an inconsistency is probable or not.

To exemplify this method of reasoning about inconsistency, re-consider the example from figure 8. Ultimately, the reason that an inconsistency is present is entailed from the observation that the values assigned to the property *"age"* differ. However, this conclusion is drawn based on the additional assumptions and arguments that any person can only have one *age*, and that, in both models, the same predicted entity (i.e., the same person) is being referred to. The latter is a result of an argument based on the premise that a person's uniqueness is identified by their name and social security number. Indeed, only if the truth of the conclusion that one and the same person is being referred to holds, can the models ever be inconsistent. This *overlap* among the models is particularly difficult to derive using just the given information and knowledge about the semantic concept of a person. Indeed, additional information may even lead one to conclude something completely different[8]. Note that this

[8]Say that, at some later point in time, the additional information was supplied that *jamesT* actually stands for *James Thomas*, and not *James Tiberius*. This may lead to the conclusion that the models do *not* talk about the same person, and that it is therefore impossible for the ages to be inconsistent. However, in this case it would actually be the social security numbers that are erroneously defined, since the *same* social security number cannot be assigned to two *different* people – a different inconsistency, but derived based on similar, abstract premises.

also shows the *non-monotonic* quality of abductive reasoning.

A prerequisite to a computational method for abductive reasoning over a set of formal models is a mechanism for extracting the relevant information (observations) from the models under consideration. Furthermore, a mechanism for (computationally) processing these observations is required. This is in addition to the prerequisite that a language or formalism exists through which abductive arguments can be represented.

4.3.3 Inconsistency Types as Patterns

Within the scope of this research, the observations that lead to the conclusion of whether an inconsistency is present or not are considered part of a *model of an inconsistency*. Here, the term *model* is used make explicit that its application does not guarantee that the conclusion drawn from it is logically correct. That is, it is an abstract representation of an inconsistency, which is used in *analyzing* sets of formal models for (probable) inconsistencies.

In section 4.2.3, a classification of inconsistencies is presented. This classification was constructed on the premise that there exists a related set of recurring features that leads one to conclude that a particular type of inconsistency is present. Therefore, it is postulated that models of specific instances of inconsistencies (i.e., a set of statements about one or more entities, used in concluding their inconsistency or non-inconsistency) can be abstracted as *patterns* which quantify over certain classes of objects. Per their classical definition, *"patterns provide proven [sic] solutions to recurring problems [sic] in a specific context"* [3]. Here, patterns are used as (proven or accepted) solutions to deducing the inconsistency of a particular set of entities. The context in which the pattern applies is defined as part of the model of an inconsistency.

It is hypothesized that, for each *type of inconsistency*, a related pattern can be identified that acts as a basis for (abductively) concluding that an inconsistency is likely to be present in a particular (set of) formal model(s). A prerequisite for evaluating the validity of this hypothesis (quantitatively) requires a mechanism for extracting the relevant observations from the set of models under consideration, and in a representation that a (computational) entity is capable of interpreting and processing so that an abductive argument can be formulated.

Note that the use of patterns to identify inconsistencies does not rely on the properties of completeness or consistency of an underlying formal system. A set of inconsistency patterns can be seen as a *partial* definition of a formal system, since these allow one to identify models that are *not* a theorem. A set of inconsistency patterns is *complete* exactly if it is able to discover all non-theorems of a formal system. A set of inconsistency patterns is *consistent* if their application does not lead to contradictory conclusions (i.e., an expression *is* inconsistent, and is *not* inconsistent). However, by definition of inconsistency management, a set of inconsistency patterns must not be complete. *How* complete a set of inconsistency patterns is, is a question of *value*, and is considered outside the scope of this dissertation.

Using an abductive approach to reasoning about inconsistencies using patterns has an additional interesting quality: for finite models, a *check* for an inconsistency (i.e., an application of the pattern) will *always* terminate. That is, for any input, an answer can always be computed. However, depending on the concrete data structure, algorithms and assumptions, it may be *intractable*[9]. Note the similarity of the approach to rule-based approaches from the related literature (see section 3.4.3). In rule-based approaches, matching an antecedent pattern of a rule is sufficient evidence to conclude that an inconsistency is present. Here, a similar argument is made, with additional observation and acknowledgment that the conclusion drawn from matching

[9]Intractability refers to problems that are theoretically solvable (given large, finite time), but any (known, best) algorithmic implementation terminates only after a very long time (where *very long* refers to *too long for practical applications*).

such a pattern is not *guaranteed* to logically entail an inconsistency.

4.3.4 Detecting Semantic Overlap by Analyzing Similarity

An essential premise on which the arguments for inconsistency of each of the examples from figure 7 to 9 are based is the existence of an *overlap*. In some cases, this overlap is explicitly stated as part of a model (e.g., in the example in figure 7, it may be assumed that the *instance of* relations are explicitly known), in which an argument for its existence is trivial. However, in other cases, such as in the example from figure 8, the overlap is not explicitly stated, and also not obvious. In fact, as argued in the previous section, the overlap cannot be logically entailed. Therefore, within the scope of the presented framework, semantic overlap is identified by abductive means whenever it cannot be logically entailed (e.g., using explicit knowledge of these).

Abductive reasoning about (semantic) overlap is non-trivial. In the related literature on ontology and database schema matching, inexact approaches to reasoning about such overlap is typically performed by measuring the similarity (sometimes referred to as *semantic distance*) [59] of certain properties of the involved entities. This similarity is then used for the purpose of formulating an argument about how likely the overlap is. Commonly used as a property for similarity is the identifier of an element – i.e., its name. This is typically based on string similarity measurements such as the *Levenshtein distance* [137] or *Hamming distance* [90], which are measures of how many operations must be performed on a string to convert one to the other. Typically, similarity of the semantic context is also measured. For instance, in a modeling language that supports the concept of *part-whole* relations, the similarity of the name of the owner is commonly accounted for as well. Other known semantic relations, such as *hyponymous* (*type of*), may also be considered. Other common measurements of semantic similarity are (similarly) of a topological nature. A variety of similarity measurements are typically combined (e.g., as a weighted sum) to determine the how likely it is for two given expressions to overlap (semantically). The application of a number of commonly used similarity measurements for ontology matching can be found in, e.g., [60].

Within the context of this research, arguments about a possible model overlap are utilized as intermediate arguments in support (or opposition) of the conclusion that a particular type of inconsistency is present. Therefore, similarity measurements, and patterns of expressions suggesting the presence of a (semantic) overlap are considered a part of the model of an inconsistency.

4.3.5 Learning from Experience

Abduction is, as recognized previously, a reasoning process that is not guaranteed to lead to a (logically) correct outcome. In practice, it may very well be that important *evidence* (i.e., an explanatory reason) to suggest the presence of an inconsistency in a particular context is not considered, or the impact of such evidence (i.e., its strength) is misjudged. In such cases, the corresponding model of an inconsistency should be *revised*. Being based on the observation that unfavorable results were achieved (e.g., too many wrong conclusions were drawn), a revised model of the inconsistency that results in improved results is said to be the result of having *learned* from experience.

Learning is an important part of inexact reasoning approaches, since it can influence the accuracy of the conclusions positively. Within the context of this framework, learning is understood as both a manual and a computational process. The latter is a classical method from the domain of *machine learning*.

4.4 Summary

In this chapter, fundamentals of inconsistency and inconsistency management are presented. Inconsistency management is defined as the discipline of identifying and resolving inconsistencies in models. Inconsistency itself is defined as *"a state of conflict in which, under an interpretation, two or more related statements are accepted* *as true, that cannot jointly be true. Given the information and knowledge available for reasoning, and a consistent interpretation, sufficient evidence exists to conclude that no situation exists in which the particular set of statements under consideration can be true".*

In the first part of the chapter, the value (i.e., utility) of inconsistency management is defended by showing that, in general, it is impossible to prove consistency. Implications on MBSE of these insights, initially derived from formal language theory and automated proof mechanisms, are then outlined in detail. For instance, characteristics and properties that result from composing formal models are presented in great detail.

The definition of the term inconsistency is derived based on a review of existing definitions, and by analyzing the characteristics of practical inconsistencies in models. For the latter, a number of examples of inconsistencies are presented. In addition to being characterized, a classification of inconsistencies is presented. In this classification, a differentiation is made between *strong* and *weak* inconsistencies, the latter of which forms an open set.

Finally, a framework for inconsistency identification is presented. Abductive reasoning is proposed as a means to derive possible model overlap and inconsistencies. Due to the nature of abductive reasoning, the use of machine learning techniques is introduced as a possible strategy for improving results over time.

CHAPTER V

REPRESENTING & REASONING OVER HETEROGENEOUS MODELS

In this chapter, a conceptual basis for representing, performing symbol manipulation in, and reasoning over heterogeneous models is presented. Model heterogeneity manifests itself in three dimensions: firstly, the different *types* of models and their *nature* (specification or analysis, process or artifact) that are used in each domain; secondly, incompatibility of meta-models; and thirdly, an extensive tool landscape with very limited integration. The concepts developed in this chapter constitute the necessary fundamental basis for a probabilistic reasoning framework in Model-Based Systems Engineering, as it enables the analysis of engineering models by computational means regardless of their corresponding language, formalism or nature. The goal of the chapter is to answer, in part, research questions 2 and 3, and to investigate hypothesis 3.

A common formalism, which is necessary for symbolic processing across heterogeneous models, is derived in the first part of the chapter. This common formalism is based on directed, labeled multi-graphs. Methods for retrieving and manipulating information and knowledge from models represented in this common representational formalism are introduced and formalized in section 5.3. This is followed by section 5.4, in which methods for performing reasoning and inference in the common formalism are introduced. Finally, in section 5.5, a semantic abstraction mechanism enabling higher-level reasoning is introduced. The chapter closes with a summary of the developed concepts.

5.1 Data, Information & Knowledge and Formal Models

As established in chapter 4, a prerequisite to identifying inconsistencies in models is an ability to interpret and analyze models. This requires a mechanism for symbolic manipulation in models, which is non-trivial to define for the case of heterogeneous models. In this section, the foundations for a common, unifying formalism allowing symbolic processing across heterogeneous models are laid.

5.1.1 Processing and Reasoning with Symbolic Expressions

As introduced in section 2.1.1.1, symbols are elements of formal languages. Symbols are used as marks that stand for or represent something [203]. Therefore, symbols can represent objects, qualities, processes or quantities from various domains. For example, in section 2.1.1.1 alphanumeric characters (the set of which is called the alphabet) were concatenated to form words, formulas and sentences. On the other hand, in section 2.1.2 lines, arcs and text (all different kinds of markings) were used to form symbols and, ultimately, utterances of the language. The issue of what *markings* constitute a symbol is intimately bound up in how a person or machine *recognizes* it. Saying that something is a symbol implies the choice of a recognizer.

Computers are designed to process symbols and expressions (i.e., arrangements of symbols) stored in memory. The symbols stored in memory can be recognized by the machine. However, without an *interpretor* (and only a recognizer) no context is provided, and the symbols and expressions are mere *data*. Therefore, data are recorded (captured and stored) symbols without context [140]. When data are processed or analyzed (e.g., by a computer program), they become *information*. Information is a *message*, which contains relevant meaning, implication, or input for a decision or action [140]. How different pieces of information relate to one another is known as *knowledge*. Knowledge is the cognition or recognition (know-what), capacity to act (know-how) and understanding (know-why).

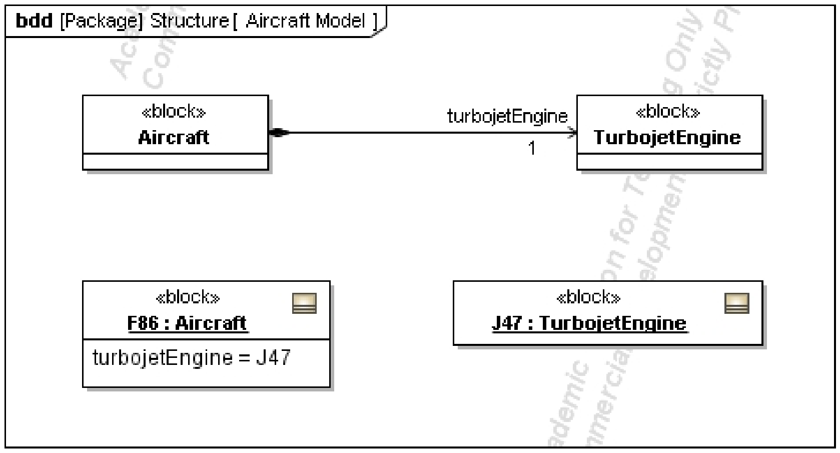


Figure 11: Sample OMG SysML [160] model (block definition diagram) illustrating the encoding of information in formal models.

To elaborate on the notion of data, information and knowledge and their relations to symbols, consider the following example: say one is given the expressions "F86" and "J47". These expressions are meaningless data without context. However, messages are formed by adding context and stating, e.g., that *F86 is an aircraft*, and *J47 is a turbojet engine*. This also adds semantics. Given relevant knowledge, one can then form the additional piece of information that links these two expressions through a relationship: *the engine type of the F86 aircraft is the J47 turbojet engine*.

The ability to interpret expressions allows one to do various kinds of reasoning on the information and knowledge encoded using expressions. Typically, computational systems that allow for such interpretation have an ability to *retrieve*, *compare* and *write* symbols. In section 2.2, a logical formalism was used to explain such automated reasoning processes using deductive techniques. There, information and knowledge are encoded as propositions using (compounded) predicated statements and implications. These are ways of encoding information in a formal language (rather than natural language), and the expressions are represented and structured in such a way that a particular interpretor understanding this formal language can process them.

Logics are not the only formalisms through which information and knowledge can be captured: (formal) models also encode information and knowledge. Consider the SysML (*Systems Modeling Language* [160, 72]) block definition diagram from figure 11. Given appropriate means to interpret the symbols represented on the diagram (i.e., an understanding of the syntax and (at least to some degree) semantics), statements very similar to those given in the previous example can be extracted. For example, *F86 is an instance of (a type of) aircraft, the latter being denoted by a type of SysML block that is labeled "Aircraft".* The fact that this is possible should not come as a surprise, since models are used for the purpose of communication. This includes making statements, as well as encoding information and knowledge about a particular system that is to reside and be deployed in a particular environment [72, 26]. For this purpose, models have a defined structure for organizing the relationships between, and contextualizing data.

5.1.2 Interpreting & Reasoning over Heterogeneous Models

Textual statements and iconic diagrams are two possible ways of capturing and representing information and knowledge. Given appropriate means to extract the relevant information and knowledge, and interpret the expressions, reasoning can be done. As has been established in chapter 4 and section 3.4, both a recognition and an interpretation mechanism are necessary for identifying inconsistencies. However, the fact that several representations for information and knowledge are used in a typical systems engineering or design scenario brings about additional challenges. Primarily, the challenge lies in the fact that multiple interpretors are required which must also interplay.

In practice, this has led to ad hoc systems such as tool chains or point-to-point model transformations to be implemented [24]. In both cases, the information and knowledge contained in the involved models is typically processed using procedural code. A much more desirable, but also more complex approach to addressing this issue is the use of a *common* representation (and interpretation) technique, and the transformation from the various formalism to such a common formalism. Modern examples of this are the *Generic Modeling Environment* (GME) [136], Soley (booggie) [99, 100], VIATRA (and VIATRA2) [35] and the approaches mentioned in [181, 168]. These approaches have in common that an attempt is made to create a single modeling language (and formal system), using which the information and knowledge encoded in a number of other models can be represented and correctly interpreted. This model then represents the product of the composition of the various heterogeneous models. However, as discussed in section 4.1.2.2, constructing such languages and appropriate morphisms from (and to) other languages is non-trivial. It is therefore no surprise that these approaches are limited (often severely) in their representational and interpretation qualities, as well as some non-functional qualities such as visualization of the product of the composition [12].

Since the identification of inconsistencies in heterogeneous models requires the extraction and interpretation of the relevant information from a potentially very large number of models, a unified technique for representing and extracting information and knowledge from models is desirable. In addition, a way of capturing knowledge on how to process the extracted information (for the purpose of identifying inconsistencies) should be incorporated into the same formalism. One such possible formalism based on graphs and graph transformations is introduced in the following.

5.2 Representing Models by Graphs

In 1982, Brian Smith, a pioneer in artificial intelligence, presented his *knowledge representation hypothesis* [198]:

> *"Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process*

exhibits, and (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge."

This hypothesis underlies most modern work in artificial intelligence [138]. Granting it, a key property that must be satisfied is that it must be possible to interpret the structures that represent and encode information and knowledge as *propositions* (i.e., as truth bearing statements).

In the artificial intelligence community, particularly within the context of *expert systems* [81], automated reasoning and knowledge representations are well-studied subjects. Many forms of information and knowledge representation exist, logical formulas and graphs being the most prominent types. Well-known techniques from the literature include *logical databases*, *semantic nets*, *frames*[1] and *object-attribute-value triples* [81, 203] (see, e.g., the expert system MYCIN [194]). While these representation techniques use very different symbols, all can be understood to encode information and knowledge in some propositional form.

5.2.1 Propositional Graph Triples

Propositions can always be represented in a form that has three elemental parts: a subject, predicate and object (at least at some level of abstraction) [152]. Such structures are generally referred to as *triplets* (or *triples*). In this form, propositions represent information and knowledge, because a basic structure for organizing the relationships (using predicates) between different subjects (and objects) exists. This is similar in spirit to object-attribute-value triples.

If a subject-predicate-object triplet is represented by two vertices (one each for the subject and object) and these vertices are connected by a directed edge (to indicate

[1]Object-oriented modeling languages such as OMG UML and OMG SysML can be interpreted as being based on the frames paradigm.

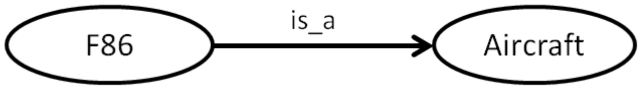


Figure 12: Subject-predicate-object triplet denoting a proposition, represented by a graph.

the predicate, and differentiate between subject and object), it is only natural to think of a triplet as a *graph*. Graphs can be defined as follows (note that the definitions of graphs that follow are loosely based on definition from [231, 184]):

Definition 5.1. *A simple directed graph is a tuple $\boldsymbol{G} = (V, E)$, where $V = \{v_1, ..., v_n\}$ is a finite set of vertices, and E is a set of tuples over the relation $E \subseteq V \times V$. E is a set of ordered pairs (v_i, v_j), each denoting an edge from a vertex v_i to a vertex v_j.*

The graph representing the proposition encoded in figure 11 has two vertices and one edge. Formally, this graph is $G_{S1} = (V_{S1}, E_{S1})$ where $V_{S1} = \{``F86'', ``Aircraft''\}$ and $E_{S1} = \{\text{is\_a}\}$, and where $\text{is\_a} = (``F86'', ``Aircraft'')$. For simplicity, the vertices are defined as (and identified by) strings of symbols, and a shorthand label is used as an identifier for the edge.

Capturing propositions in graph form is similar to the idea behind semantic nets (sometimes referred to as *propositional nets*) [152, 176]. In figure 12, the proposition *F86 is a type of Aircraft*, denoted as the triplet `(F86, is_a, Aircraft)`[2], is represented by a graph triplet.

Note that even simple statements asserting a state or quality of a subject can be represented in such a manner. For instance, consider a proposition that states that the F86 aircraft flies. One way of representing this statement is: *F86 flies*. However, given that "flies" is an action (or perhaps state, depending on the context and information available), the proposition can also be represented as *F86 performs_action flies*, where *performs_action* is the predicate. In a similar manner, complex, interlinked statements

[2]In a predicate logic, this proposition would typically be represented as a predicated statement in the following form: `is_a(F86, Aircraft)`.

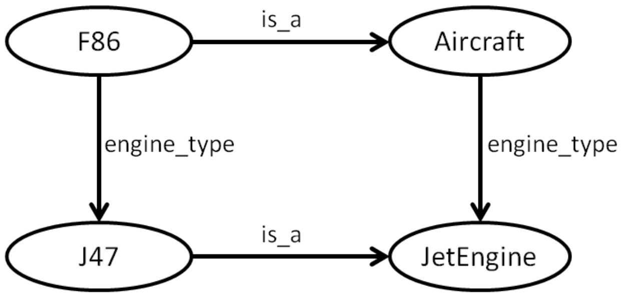


Figure 13: Complex, interlinked set of propositions represented by a graph.

can be represented by breaking a larger proposition into smaller ones and relating them with predicates. For instance, in figure 13, the information *The F86, which is a type of Aircraft with a Jet Engine, has a Jet Engine of type J47* is divided up into a set of four propositions, which are represented by a single graph. Notice how multiple statements are made about the subject “F86”. Also, “J47” acts as a subject in one proposition (*J47 is a type of Jet Engine*), and as the object in another (*F86 has engine of type J47*).

5.2.2 Directed, Labeled Multi-Graphs

In the given examples (see figures 11 and 13)), vertices and edges were identified by simple strings as labels. However, names are not always appropriate means for uniquely identifying objects, but should be considered attributes of vertices and edges. Therefore, definition 5.1 is expanded upon to incorporate the more generic notion of *labels* for vertices and edges, which provides a basis for attribution. In addition, concepts from formal languages from section 2.1 are incorporated.

To allow for labels in a directed graph, the use of directed, labeled multi-graphs [231] is proposed as a means for meaningfully capturing information and knowledge. Here, the term multi-graph indicates that the graph may contain cycles. Similar to the classical definition of formal languages (see section 2.1.1.1, an alphabet is used as a basis for forming terms (here: graphs). Let Σ be such an alphabet, and let Σ_N represent the subset that contains only the non-logical constants of Σ. Directed and labeled multi-graphs over an alphabet of non-logical constants Σ_N can be defined as follows:

Definition 5.2. *A directed and labeled multi-graph over an alphabet* Σ_N *is a tuple* $\boldsymbol{G} = (V, E, e, L, l_V)$*, where* $V = \{v_1, v_2, ..., v_n\}$ *is a finite set of vertices, and* $E = \{e_1, e_2, ..., e_m\}$ *is a multiset denoting edges, where each edge* $e_i \in E : e_i \in V \times V$ *is defined as a tuple (ordered pair) of vertices, the first element denoting the source and the second element the target vertex for a directed edge. The partial function* $e : E \to V$ *assigns edge definitions to vertices.* $L \subseteq \Sigma_N^*$ *is a set of labels and* $l_V : V \to L$ *a partial function that assigns vertices to labels.*

Definition 5.2 has been inspired by the definition of *E-Graphs* given in [58], and is an expanded version of the definition previously published by Herzig and Paredis in [106]. In the definition, graphs have vertices, which are connected by edges. These edges are captured as ordered *pairs*, where the first element represents the source vertex, and the second element the target vertex. Vertices can be associated with a label which is, similar to the definition of terms, an element of the set of all combinations of symbols from the alphabet used. Edges are given a *definition* by associating each edge with a particular vertex (where one vertex can act as the definition for more than one edge). This gives edges an identity and allows for more complex definitions of edges (and is similar to a *typing system* for edges), since vertices defining edges can have relations to other vertices.

A sample graph constructed following definition 5.2 is visualized in figure 14. Note that, in the following, the shorthand notation is used as illustrated previously in figure 11, which eliminates the edges that indicate assignments of labels to vertices and edges to vertices, and shows the labels associated with vertices rather than the vertex number (similar for edges, where the label of the vertex associated with the edge is used as an edge label).

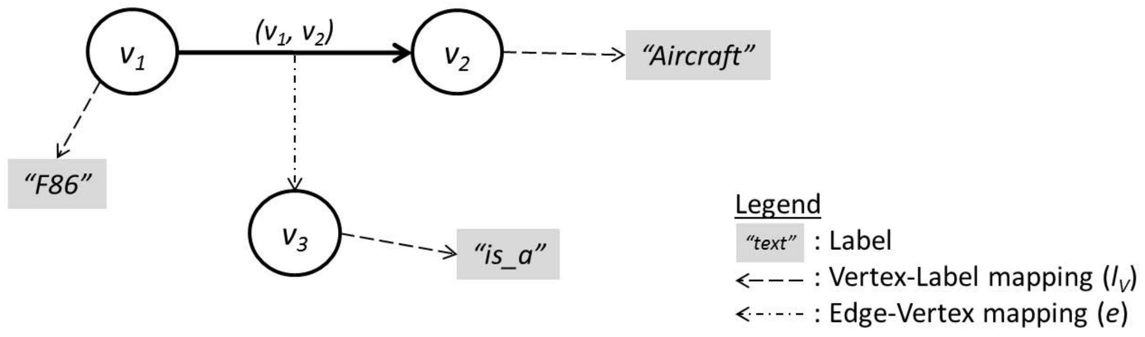


Figure 14: Sample graph visualizing the given definition of directed, labeled multigraphs.

5.2.3 Formal Models as Propositional Graphs

Formal models are typically created for the purpose of capturing statements about a particular system. These statements are *factual* (i.e., facts) in that their truth value is known. Therefore, to represent a proposition by a graph triple, it is sufficient to capture it in subject-predicate-object form, assuming that the truth value of the statement is always `true` unless proven otherwise. These factual statements are of primary interest for the purpose of identifying inconsistencies.

The agglomeration of all statements made (i.e., the information and knowledge available about a system) should be representable by a graph. This leads to the following proposition:

Proposition 5.1. *At some level of abstraction, the information and knowledge encoded in any model can be represented by a graph.*

There are a number of reasons why the representation of models (and, hence, also propositions) by graphs is advantageous. For one, using graph-based representations of propositions extracted from models can be more intuitive than a representation in a textual language [91]. This is because complex relations can be extracted more easily, for instance, when collecting all statements made about a particular subject. Also, graphs are mathematically elegant and formal structures, and the retrieval and manipulation of graphs is well-studied [184]. As discussed in section 2.1, both syntax and semantics can be defined using *graph transformations* (i.e., operations on graphs that modify a source graph).

The validity and viability of proposition 5.1 is strongly supported by the fact that (as discussed in section 2.1.2) the meta-model of any model can be represented by a graph (recall that this is referred to as the *type graph*) [82], and by the generally accepted fact that for all formal languages a meta-model is definable (or identifiable). Therefore, an instance of a graph-based meta-model (i.e., a model) must also be representable by a graph. Note that a meta-model is a model that makes statements about the *language*, while the model makes statements about an *utterance* of the language. For instance, the meta-model of UML makes statements about entities such as *UML Classes* and *UML Properties*, while a UML model makes statements about instances of these entities – i.e., entities related to, e.g., a particular software system.

In the related literature, the representation of models by graphs is not uncommon. As discussed in chapter 3, VIATRA (and VIATRA2) is an example of a model-based development framework and tool that supports graph based representations of models. VIATRA2 also supports the importing of models based on other standard meta-modeling cores [9] such as *OMG Meta-Object Facility* (MOF) [163]. This supports the viability and practicality of graph-based models.

5.2.4 Transformation & Interpretation

A graph-based representation of a model can be constructed by defining a morphism $g : \mathbf{M} \rightarrow \mathbf{G}$ between a model $\mathbf{M}$ and a corresponding graph representation of the model $\mathbf{G}$. In the limit, such a morphism is an *isomorphism.* That is, every expression from a formal modeling language is translated to a graph-based form. For formal modeling languages, this includes both syntax and semantics, where syntactic expressions can be translated to static graphs, and the semantics (including structural well-formedness constraints) to a series of graph transformation rules (which conform to some standard graph transformation language with high expressivity for which an interpretor exists).

For the purpose of identifying inconsistencies, a full translation of the language-specific semantics is not always meaningful. This has mostly practical reasons, in that most modeling tools have the ability to check, e.g., the syntactical well-formedness of a model. In addition, a complete translation of the semantics (semantic domain and semantic mapping) requires a graph transformation formalism to be available, which must possess sufficient expressiveness to represent every type of semantics (e.g., execution semantics). Therefore, the translation of semantics considered in the following is not claimed to be complete.

As argued in section 4.3, inconsistencies can be discovered through the application of negative constraints, i.e., through the matching of patterns. Since such pattern matches are purely syntactical, the focus set in the following is on translating syntactical expressions. Therefore, most *semantic inconsistencies* are assumed to be identifiable from the syntactic context provided.

5.2.4.1 Choosing a Level of Abstraction

Morphisms for transformations from a model to a graph-based representation are not restricted to isomorphisms (such as is common in model-to-model transformation scenarios). That is, not all of the contained propositions (i.e., information associated with a model) should necessarily be translated to the target formalism. Non-isomorphic morphisms can result in the loss of information. However, for the purpose of *identifying inconsistencies*, translating *all* information and knowledge contained in a model may not always be meaningful. When thinking of inconsistencies as patterns (such as was done in section 4.3), only the information that can trigger a match to a pattern (i.e., that could, potentially, be part of a match to a pattern) is of value. Theoretically, any information and knowledge that is not a part of any match to a part of a pattern (representing a type of inconsistency) can be disregarded.

To give a concrete example, consider the case where the only inconsistency of interest is a mismatch between a part hierarchy in a CAD model and a block hierarchy in a SysML model. In this case, information such as that related to polygons and their spacial location is not of interest for the particular case at hand. However, including the information would not *harm* the identification of an inconsistency – it would merely provide a larger set of propositions as a basis for reasoning, most of which will never be relevant for identifying inconsistencies.

5.2.4.2 Syntactic Transformation

The transformation of a formal model to a graph-based representation follows a simple pattern: for every relevant element (which includes concepts, relations, individuals, classes and other syntactic entities used in modeling languages) a vertex is created, which acts as the definition for the particular element. Every element that is related explicitly to another element in some form (including known *type* relations such as *instance-of*) is then indicated by an edge between the vertices defining these elements, where bidirectional relations are made possible through creating two directed edges. These edges are then mapped to the vertices defining the edges. Note that this transformation results in a graph where the various meta-levels are mixed and, without an appropriate interpretation, the different meta-levels are (syntactically) not differentiable from one another.

To strengthen the understanding of this transformation pattern, consider the case where an isomorphic image of a model and its meta-model are to be translated[3] (including the instance relationships between the two models). Furthermore, assume knowledge of the meta-meta-model. Also assume that this meta-meta-model is that for class diagrams (which, as established in section 2.1 is the de-facto standard for meta-modeling). Every meta-meta-model class, and every meta-meta-model relationship is represented by a vertex with the label corresponding to the name of the corresponding element. Meta-models and models are translated in a similar fashion. The concept of *instance-of* relations is translated using the corresponding language-specific vocabulary to a vertex, which is given the appropriate label representing this construct, and edges semantically representing these relations are associated with the vertex. The process is analogous for other relations.

Note that the transformation to the provided formalism expects a graph-like structure of the underlying model to be present (or at least formable). However, as mentioned in section 2.1, this is, at least in theory, always possible.

5.2.4.3 Transformation of Semantics & Interpretation

Given that a reference exists between a syntactic expression in a formal modeling language and the corresponding generated graph-based representation, the semantics of the graph-based representation can simply be defined in a translational manner. Whenever this is not possible (or not practical) certain flavors of semantics can be translated to a graph-based representation by translating interpretative functions to graph transformation rules. However, this, in turn, requires a language for expressing these graph transformations, which must be expressive enough for the purpose of translating the semantics and for which an interpretor must exist.

For the purpose of identifying inconsistencies, the (often language-specific) semantics relevant to identifying inconsistencies (i.e., that affect related patterns) should

[3]Whether the required information is extracted through a tool API, or is made available in some other form is considered an implementation-specific issue.

always be translated. This includes commonly encountered language semantics for relations, which include *transitivity*, *reflexivity* and *symmetry*.

5.2.4.4 Interpretation of Compositions of Graph-Based Models

When translating a single model to a graph-based representation, the semantics of the graph-based model are, as explained in the previous section, simply translational. However, the semantics of the graph-based model become non-trivial when multiple, heterogeneous models are transformed, and the results of all transformations to the graph-based formalism are considered as one graph. In such a case, the formal system of which (at least implicitly) the graph-based model is a part, and those from each individual translated model, are *no longer equivalent.* This is because of the disparity and heterogeneity of the models, and (as often encountered in practice) lack of explicit knowledge about the relations between the models (i.e., the model overlap, which is primarily of a semantic nature).

Identifying a semantic domain and semantic mapping for such a set of composed graph-based models is non-trivial, and, at least in practice, a universal semantic domain is unlikely to be definable due to the associated complexity. In theory, such a universal domain would require the ability to explain all phenomena with perfect precision, which is unlikely, particularly due to the findings from section 4.1.2 and as published by Herzig *et al.* in [102]. The approach for identifying inconsistencies introduced in section 4.3 can be understood as an enabler for a partial check for (in)consistency using an *incomplete definition of a formal system.* Contrary to the approach outlined in section 2.2, the approach taken in this research (which is first introduced in section 4.3) is not to prove the *consistency* of an expression (i.e., a subgraph), but prove its *inconsistency.* Therefore, the formal system is defined only partially by a limited set of axioms and inference rules, and only certain non-conformances of a statement to the formal system can be proven.

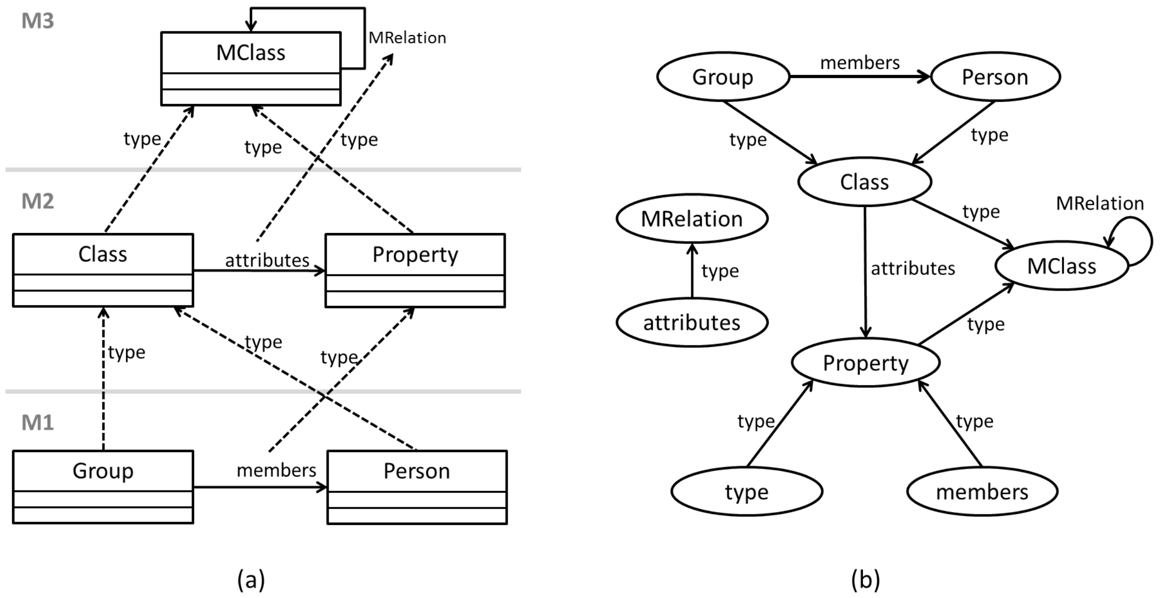


Figure 15: (a) Example model, meta-model and meta-meta-model defined using a (simplified and hypothetical) class diagram syntax, and (b) the corresponding graph-based representation.

5.2.5 Example: Translating a Multi-Level Model with a Class Diagram Like Syntax

To exemplify the translation of a formal model to the introduced graph-based formalism, a hypothetical model and its meta-model and meta-meta-model are considered (see figure 15a). It is assumed that the formal model to be translated to the graph-based formalism has been created using a class diagram-like formalism, where the meta-meta-model is defined to be reflective (i.e., self-describing) similar to the *Meta-Object Facility* (MOF). Note that the example does not discuss any concrete serialization of the graph-based model, but is merely meant to serve as an illustration example of the proposed procedure of constructing a graph-based model.

The graph-based representation of the model depicted in figure 15a is depicted in figure 15b, with the translation defined in accordance with the schema defined in section 5.2.4.2. In the example, both the notion of an element (`MClass`, `Class`, `Property`, `Group` and `Person`) and that of a relation between such elements (`MRelation`, `type`, `attributes`, `members`) exists. As outlined in section 5.2.4.2, for each of these elements and relations a corresponding vertex is created (for semantically equivalent elements or relations only *one* vertex is created). The syntactical names used in the source model are added to the set of labels, and the mappings from vertices to labels are defined (which, by the introduced convention, are not shown). Thereafter, for each relation, a corresponding edge is created between the affected elements. Mappings are then defined between these newly created edges and the relevant vertices (similar to the mappings of vertices to labels, these mappings are, by the introduced convention, not shown).

In order for the transformation to be possible, a mechanism must exist through which the information in the given model can be extracted. In a practical application, this extraction mechanism is enabled by an *application programming interface* (API), or through parsing and interpretation of a serialization of the model in a well-defined format (e.g., using the *extensible markup language* (XML)). The latter is common in model transformation environments where languages such as QVTO [162] aid in defining the transformation, while taking care of the parsing of a (standardized) serialization of the model in the background (see, e.g., [109] for an example).

Note that *what* information is transformed to the graph-based representation is, as discussed before, arbitrary to some degree and defined by the morphism. However, at least all information relevant to identifying particular types of inconsistencies should be transformed. In the limit, all information and knowledge extractable from a model should be transformed. In this case, all of the *abstract* syntactical information was extracted.

As outlined in section 5.2.4.2, following this transformation scheme leads to a representation of a model in which the various meta-layers are no longer clearly distinguishable without an interpretation of the graph. That is, without knowledge of the semantics of the `type` relationship, the meta-levels are not distinguishable.

5.3 Querying Graph-Based Models

In the previous section a model for representing the (syntactic) information and knowledge contained in heterogeneous formal models in a common, graph-based formalism is introduced. The result is a static and syntactic translation of a model. However, in order to interpret the constructed graph-based model, mechanisms for extracting and manipulating information and knowledge contained in a graph-based model are needed. Such mechanisms are enabled by posing *queries* and executing *transformation rules*, the basis of which is *pattern matching.* A pattern matching formalism and a mechanism for *retrieving* information from a graph-based model is introduced in the following.

5.3.1 A Graph Pattern Formalism

Patterns are utilized for the purpose of locating information in a constant structure [203]. Here, this constant structure is the *data graph* considered – i.e., the graph-based representation of a set of formal models. Hence, a pattern identifies one or more *subgraphs.* Key to the definition of patterns are the concepts of *variables* and *constants* (see section 2.2, where it was identified that alphabets consist of constants (logical and non-logical) and variables). Hence, the definition of a formalism for defining patterns should, at minimum, define symbols for both variables and constants. Information is located by (syntactically) matching constants defined in a pattern to corresponding constants in a data graph. Pattern variables act as *wildcards*[4] and can be *bound* to any matching vertices (or edges).

Within the context of this dissertation, patterns which incorporate only constants and variables are referred to as *simple patterns.* To define the notion of simple patterns, let Σ be the same alphabet as used in the definition for directed, labeled

[4]A wildcard is a symbol that may be substituted for any other symbol from a defined set (here, the set of non-logical constants).

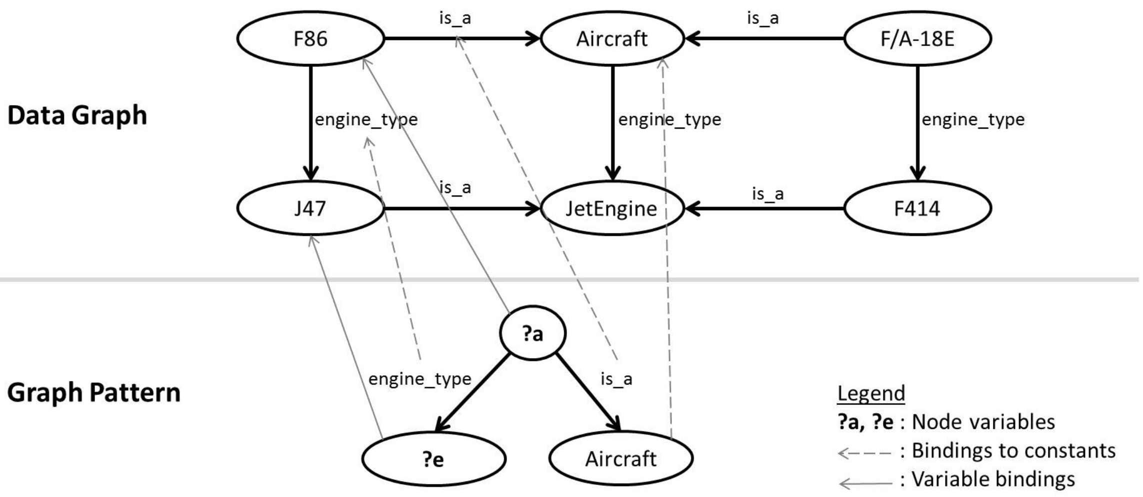


Figure 16: Example pattern, and bindings of constants and variables in the graph pattern to a data graph.

multi-graphs (see definition 5.2). Furthermore, let $\Sigma_{var} \subseteq \Sigma$ and $\Sigma_N \subseteq \Sigma$, where Σ_N is the subset of non-logical constants over Σ and Σ_{var} the set of variables, where Σ_N and Σ_{var} are disjoint. Simple graph patterns are then defined as:

Definition 5.3. *A simple graph pattern $\boldsymbol{P}_S$ is a directed, labeled multi-graph $\boldsymbol{P}_S = (V_P, E_P, e_P, L_P, l_{V_P})$ with $V_P = V_N \cup V_{var}$ as the set of vertices composed of the two disjoint sets V_N and V_{var}. V_N represents the set of vertices with constant labels and V_{var} the set of vertices denoting variables. The set of edges is defined by E_P, which are assigned vertices through the function e_P. L_P is the set of labels defined by $L \subseteq \Sigma_N^* \cup \Sigma_{var}$. Vertices are assigned labels through the mapping l_{V_P}.*

Similar to the definition of the target data graph from which information is to be retrieved, patterns are defined as labeled, directed subgraphs. However, their definition is extended by allowing for both non-logical constants and symbols denoting variables as labels (by definition of the partial function l_{V_P}, any vertex can at most be assigned one label and can hence represent only either a constant or a variable). Matching simple patterns is done by mapping from the data graph to the elements with constant labels while treating variables as wildcards.

Definition 5.3 is an enhancement of a previously published definition by Herzig and Paredis [106], originally inspired by the definition in [76]. An example of a *simple graph pattern* is given in figure 16, where a pattern is defined with two variables (**?a** and **?e**) and three constants (*"engine_type"*, *"is_a"* and *"Aircraft"*). One set of possible bindings to elements from the target data graph is illustrated.

Using only variables and constants in defining patterns limits their expressiveness. In practice, this is typically mitigated by incorporating logical operators and functors, which are simply functions evaluated in addition to matching the simple pattern that, when interpreted, return either `TRUE` or `FALSE`. For this purpose, the notion of *simple graph patterns* is extended in definition 5.4 to that of *complex graph patterns*.

To define complex graph patterns formally, (a subset of) the logical constants from the alphabet Σ are now considered as well. Let $\Sigma_L \subset \Sigma$ be a set of logical constants that is disjoint from Σ_N and Σ_{var}, and which represents a set of *functors*. Complex graph patterns can then be defined in the following way:

Definition 5.4. *A complex pattern $\boldsymbol{P}$ is a tuple $\boldsymbol{P} = (\boldsymbol{P}_S, C)$ where $\boldsymbol{P}_S$ is a simple pattern and C is a labeled hypergraph $C = (V_C, E_C, A, a, L_C, l_C)$. V_C is a set of vertices for which $V_P \subseteq V_C$ (where V_P is from $\boldsymbol{P}_S$) and E_C a set of hyperedges defined by the power set of $V_C \cup E$, where E is the set of possible edges from the definition of $\boldsymbol{P}_S$: i.e., $E_C \subseteq (V_C \cup E)^2$. A is a set of tuples denoting possible function arguments, and is defined as a subset of the union of all possible Cartesian products over E_C: $A \subseteq \bigcup_{i=1}^{n} \left(\times_{j=1}^{i} E_C\right)$, where $n = |V_C|$. vertices are assigned arguments through the partial function $a : (V_C \setminus V_P) \to A$. L_C is a set of labels defined by $L_C \subseteq \Sigma_L$. The labeling function l_C is a partial function assigning functor labels to vertices from the set $V_C \setminus V_P$ and is defined by $l_C : (V_C \setminus V_P) \to L_C$.*

In definition 5.4, functors are defined as vertices in a graph with labels that represent logical constants. For each functor, zero or more *arguments* are defined. The number of arguments defines the *arity* of a functor. For instance, in the example

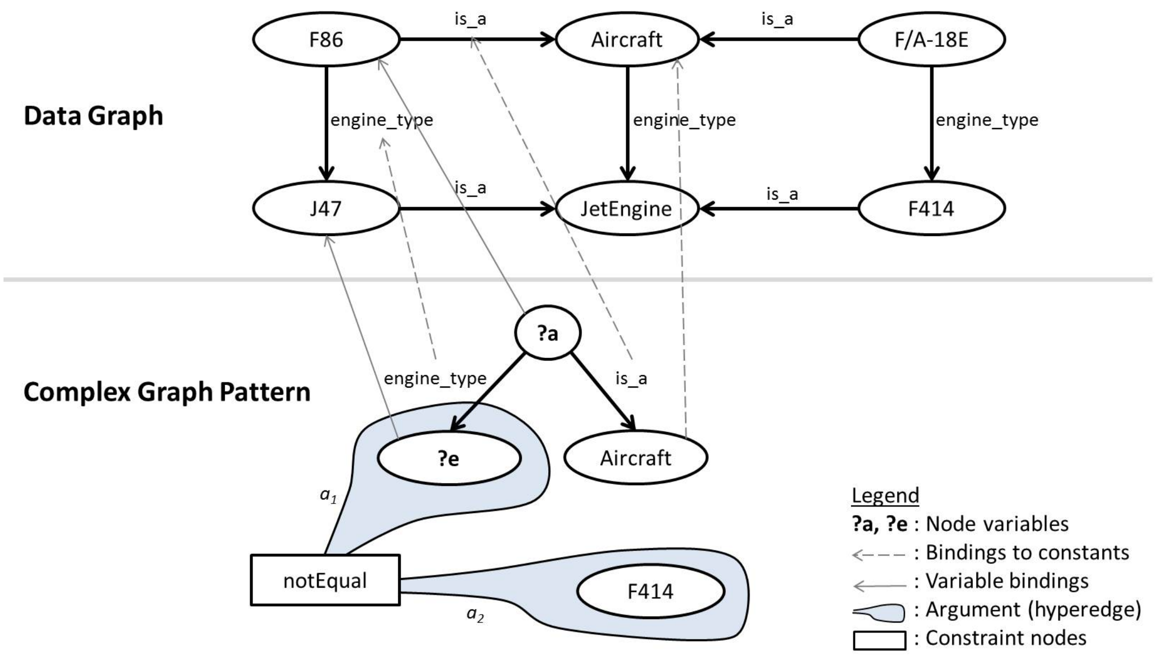


Figure 17: Example complex pattern that is a combination of the simple pattern from figure 16 and the functor *notEqual* used for finding *Aircraft* instances that do not have the engine type *F414*. Note that in this example, the argument $(a_1, a_2) \in A$ is assigned to *notEqual*, where $a_1 = \{v_{?\mathbf{e}}\}$ and $a_2 = \{v_{F414}\}$. Hence, *notEqual* has arity 2.

given in figure 17, matches to the simple pattern defined in figure 16 are restricted to those for which the functor *notEqual* evaluates to `true` (where the semantics of *notEqual* are assumed defined). Here, semantics for *notEqual* are assumed that interpret *notEqual* as `true` for all of those cases where the vertex in the data graph bound to `?e` is not the same as the vertex with the label *"F414"*. Note that definition 5.4 allows for arbitrary subgraphs to be used as arguments for the functors, hence also allowing for functors to be defined that evaluate to true for *negative matches* – i.e., whenever a pattern is *not* found. In such cases, $\mathbf{P}_S$ would be an empty graph, and only a functor with a subgraph is defined. The semantics of functors can formally be defined by mapping to a well-defined formal system such as a logic.

In definition 5.4, *hypergraphs* [231] are used to define arguments of functors. Hypergraphs are simply graphs that allow for edges that connect more than one vertex. Arguments for functors are then tuples where each entry in a tuple represents a hyperedge. Tuples are used for this purpose since the order of arguments is, in the general case, not arbitrary (except for unary operators, or, e.g., symmetric relations). Note that by including functor vertices in the definition of the set of possible hyperedges E_C, arguments of functors may also include other functors. In figure 17, the functor *notEqual* has two arguments, each of which is a subgraph with just one vertex.

Note that a match to a simple pattern can be interpreted as the evaluation of a default functor *matches* which carries the simple pattern as its only argument and returns `TRUE` if a match to a pattern was found in a target data graph.

5.3.2 Pattern Matching in Graphs

A single match to a simple graph pattern is a *set of bindings* from vertices and edges in the simple graph pattern to vertices and edges in a target data graph. This is illustrated for one particular combination of bindings in figure 16. Vertices of the simple graph pattern that have labels consisting of constant symbols can map to vertices in the target data graph that are associated with a label that is symbolically equal. Edges are mapped in a similar fashion. If the label is not a constant, but a variable, the mapping is defined as a mapping from the variable vertex (or edge) to *all* vertices (or edges) in a target graph.

In order for a vertex-edge-vertex triple of a graph pattern to match a vertex-edge-vertex triple in the data graph, the mappings of the three graph elements must be consistent for each *tuple* of mappings. That is, even though the vertex variable **?a** in figure 16 could (if considered alone) map to all vertices of the data graph, matching the vertex-edge-vertex triple (**?a**, *is_a*, *Aircraft*) restricts the mappings for **?a** to those for which there exists an outgoing edge with the label *is_a*, which is connected to a vertex with label *Aircraft*. In the case of figure 16, **?a** could therefore map to the vertex with the label *F86* or the vertex with the label *F/A-18E*. Therefore, one must consider a number of mappings at the same time, where the mappings from the elements of the graph pattern to the elements of the data graph must be consistent. For instance, for the two example matches, the mapping of the edge labeled *is_a* in the pattern must map to the edge *is_a* in the data graph that is consistent with the other mappings (note that by definition 5.2 that both edges and vertices are *not* unique by their labels).

To define this process of pattern matching formally, a number of definitions are introduced in the following. Let m_E be a mapping from edges of the simple graph pattern to edges of the target data graph $m_E : E_P \to E$, where m_E is defined by label equality; that is, m_E is a non-surjective, non-injective function defined for those edges $(v_{Pk}, v_{Pl}) = e_P i \in E_P$ for which there exists at least one mapping for which $l_{V_P}(e_P(e_P i)) = l_V(e(e_j))$, where $e_j \in E$ are edges in the target data graph. Edges with variable symbols as labels map to *all* edges of the target data graph. Similarly, let $m_V : V_P \to V$ be a non-injective, non-surjective mapping from the vertices in the simple graph pattern $\mathbf{P}_S$ to the vertices in the target data graph. m_V is also defined by label equality, where $m_V(v_{Pk}) = v_l$ is true for all vertices in the simple graph pattern and target data graph for which $l_{V_P}(v_{Pk}) = l_V(v_l)$. Analogously, vertices in $\mathbf{P}_S$ with variable symbols as labels map to all vertices in the target data graph. Furthermore, let $T = \{(v_i, e_k, v_j) \mid v_i, v_j \in V \text{and} (v_i, v_j) = e_k \in E\}$ denote the set of triples in the target data graph and, analogously defined, T_P be the set of triples in the simple graph pattern. A match to a triple can then be defined in the following way:

Definition 5.5. *The mapping m_t representing matches to a triple $t_P \in T_P$ is defined by the tuple of mappings $m_t = (m_V, m_E, m_V)$, where m_t maps a pattern triple $t_P = (v_{Pi}, e_{Pk}, v_{Pj}) \in T_P$ to a target data graph triple $t = (v_i, e_k, v_j) \in T$ whenever $m_V(v_{Pi})$, $m_V(v_{Pj})$ and $m_V(e_{Pk})$ are defined.*

Note that m_t can be regarded as a mapping pointing from one triple in the graph pattern (e.g., (**?a**, *is_a*, *Aircraft*)) to zero or more triples in the data graph (e.g., (*F86*, *is_a*, *Aircraft*) and (*F/A-18E*, *is_a*, *Aircraft*)). Using these definitions, matching of triples can now be generalized to the case of matching sets of (related) triples (i.e., the simple graph pattern as a whole):

Definition 5.6. *A set of matches $M_{\boldsymbol{P}_S,\boldsymbol{G}}$ to a simple graph pattern $\boldsymbol{P}_S$ in a target data graph $\boldsymbol{G}$ is an exhaustive set of sets of graph triples which can be constructed from $M_{\boldsymbol{P}_S,\boldsymbol{G}} = \{T_{M_i} \mid T_{M_i} = m(T_P)\}$, where $m : T_P \to T$ is a tuple of sub-mappings $(m_{t1}, m_{t2}, ..., m_{t|T_P|})$ where, for each mapping, all sub-mappings are unique and consistent, i.e., $\forall\ t_{Pi}, t_{Pj} \mid v_{Pix} = v_{Pjy} : m_V(v_{Pix}) = m_V(v_{Pjy})$ and $\forall t_{Pi}, t_{Pj} \mid e_{Pkx} = e_{Ply} : m_E(e_{Pkx}) = m_E(e_{Ply})$.*

The uniqueness and consistency criteria ensure that the same elements from the co-domain are used across all shared vertices and edges in the pattern. That is, for the pattern in figure 16, it is ensured that across the two pattern triples (**?a**, *is_a*, *Aircraft*) and (**?a**, *engine_type*, **?e**), the same bindings for **?a** are used in one match (i.e., **?a** does not map to *F86* for the one triple and to *F/A-18E* in the other, but **?a** maps to either one).

Note that $M_{\mathbf{P}_S,\mathbf{G}}$ is a set of sets of graph triples. Together, the triples in each of the subsets of $M_{\mathbf{P}_S,\mathbf{G}}$ form a (sub-)graph of the target data graph. The above definitions are also valid for the *subgraph isomorphism* problem if a simple graph pattern is defined that has no vertices with variable symbols as labels.

In practical implementations of pattern matching procedures, the target data graph must be traversed in order to define the various mappings. In the literature, there exist a variety of algorithms to find candidate expressions for possible matching, which traverse the graph in different ways. Many of these are specifically designed and optimized for graphs with specific properties (e.g., acyclic graphs or tree structures). In [33], a number of algorithms are mentioned including the most general class of *tree search* algorithms [219, 133] (note that these algorithms do not assume that the underlying data graph has a tree like structure), where the basic idea is the iterative expansion of a initially empty partial match by adding new pairs of matched vertices. The general problem of finding a subgraph in a target data graph can be shown to be in the complexity class of NP-complete problems [231, 219].

Matching of complex patterns relies on an *interpretation* I for C. Similar to the definitions given in sections 2.1.1.4 and 2.2.2, a *valuation function* v is defined which inductively assigns values from a semantic domain $\mathcal{D}$ to vertices and edges defined by C. Here, $\{\texttt{TRUE}, \texttt{FALSE}\} \subseteq \mathcal{D}$. Let a basis for I and v be a simple logic similar to 2.2.1. For brevity, only the semantics of *top level functors* – that is, functors which are not part of an argument of other functors – are detailed. Valid top level functors must valuate to either `TRUE` or `FALSE`.

Finding matches to a complex pattern is similar to that of finding matches to simple graph patterns, but with additional constraints imposed. For each match to a simple graph pattern, each functor from C is evaluated. Only if the logical conjunction of all top level functors evaluates to `TRUE` is the subgraph considered a match. For example, in figure 17, a complex pattern is illustrated. Two matches to the simple pattern can be identified. The semantics of the (fictitious) top level functor *notEquals* dictate that if the labels of the bound vertices specified as first and second arguments are symbolically equal, the functor evaluates to `FALSE` and `TRUE` otherwise. For one of the two possible matches, in which **?e** binds to the vertex labeled *F414*, the functor *notEquals* evaluates to `FALSE` under interpretation I and the match is not considered valid.

To strengthen the understanding of how complex patterns are matched, consider the anonymous functor *matches* which evaluates to `TRUE` for each match M to the simple pattern $\mathbf{P}_S$ (and also to `TRUE` if the simple graph pattern is empty). Furthermore, assume a set of top level functors f_i with arbitrary sets of arguments A_i. Then, a match to a complex pattern is found if, after forming the logical conjunction of the various functors, the logical expression $matches(M) \vee (\bigvee_i f_i(A_i))$ evaluates to `TRUE`.

The section is concluded with the following definition for matches to complex graph patterns:

Definition 5.7. *A set of matches $M_{\boldsymbol{P},\boldsymbol{G}} = \{T_1, T_2, ..., T_n\}$ to a complex graph pattern $\boldsymbol{P}$ in a target data graph $\boldsymbol{G}$ is an exhaustive set of matches to the simple graph pattern $\boldsymbol{P}_S$ defined through $\boldsymbol{P}$ for each of which, under an interpretation I, the top level functors evaluate to* **`TRUE`***. That is, a single match to a complex pattern is a set of triples $T_i \in M_{\boldsymbol{P},\boldsymbol{G}}$ that is satisfiable under interpretation I.*

5.3.3 Formulating Queries

In the following, the notion of *queries* is introduced briefly. By definition, a query is an *inquiry* for the purpose of retrieving information. Queries are typically formulated as questions posed to an information source. For instance, one could pose the following question to the example data graph used throughout this section:

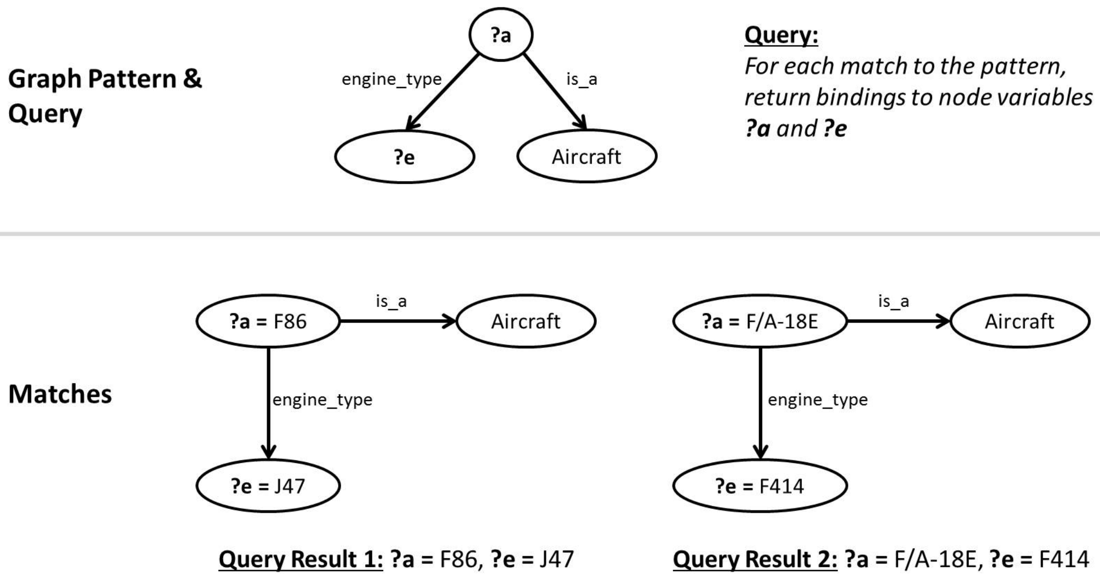

Graph Pattern & Query
?a
engine_type
is_a
?e
Aircraft
Query:
For each match to the pattern, return bindings to node variables ?a and ?e
Matches
?a = F86
is_a
Aircraft
engine_type
?e = J47
?a = F/A-18E
is_a
Aircraft
engine_type
?e = F414
Query Result 1: ?a = F86, ?e = J47
Query Result 2: ?a = F/A-18E, ?e = F414

Figure 18: Example graph pattern query with two sample query results.

“Which aircrafts have which jet engines?”

Looking at figure 16, the answer should be that *“The F86 has a J47 engine, and the F/A-18E has a F414 engine”*. Analyzing the question, one notices two things: firstly, information about the *context* of the question is given. Secondly, there are references to what information is to be *returned*. In the case of the given query, *things identified as types of aircrafts* (context), and *things identified as jet engines* (context) that are *related to the respective type of aircraft* (context) are to be returned.

To retrieve an answer computationally, the relevant information must be retrieved from the data graph. For this purpose, graph patterns can be used. In this case, the pattern defined in figure 16 can be reused and will, as discussed in the previous section, returns two subgraphs. The relevant information now needs to be extracted from the subgraphs.

Extracting information from a query computationally requires an *interpretation* of the language that the query conforms to. A formal definition of such a language for queries is not necessary for an understanding of the remainder of this dissertation and is considered outside the scope. Therefore, only a simple query mechanism is introduced, where queries, at their core, are said to consist of two parts: a part defining what should be *returned*, and a part that defines the *context* of what is to be returned. For the simple query mechanism, a (complex or simple) graph pattern with at least one variable vertex (or edge) is used in defining the context. Returned are the bindings to any number of variables from the context pattern. This pattern mechanism is illustrated in figure 18. Note that, in essence, such a query mechanism allows one to *filter* the information contained in a match to a pattern by returning only a subset of the bindings between the elements of the simple graph pattern and the data graph.

Within the scope of this dissertation only a simple query mechanism is considered, which is used to retrieve bindings to variables in matches. In the following, the notion of a simple query query and query result are defined more formally.

Definition 5.8. *A simple query is a tuple $Q = (\boldsymbol{P}, Q_V, \boldsymbol{G})$, where $\boldsymbol{P}$ is a complex graph pattern and $\boldsymbol{G}$ a target data graph. $Q_V = \{qv_1, qv_2, ..., qv_m\}$ is a subset of the vertices from $\boldsymbol{P}$ that denote the variables for which bindings should be returned.*

The execution of a query involves the matching of the associated pattern in a data graph. The result of this matching is, as introduced a set of sets of triple T_i. Here, the mapping from variable vertices of interest to vertices in the data graph are assumed preserved for each match and are depicted by the set of mappings $\mathbf{m}_v = \{m_{v_1}, m_{v_2}, ..., m_{v_k}\}$ where $k = |M_{\mathbf{P},\mathbf{G}}|$. Each mapping is defined by $m_{v_i} : Q_V \rightarrow V_{T_i}$, where V_{T_i} is defined as the set of all unique vertices in the triples $t_i \in T_i$ that represent matches to the graph pattern. A simple query result can now be formalized as:

Definition 5.9. *A simple query result is a tuple $Q_r = (Q, M_{\boldsymbol{P},\boldsymbol{G}}, \boldsymbol{m}_v)$, where Q depicts the associated query, $M_{\boldsymbol{P},\boldsymbol{G}}$ is the set of matches to the pattern defined by the query and $\boldsymbol{m}_v$ is an ordered set of mappings from variable vertices defined in the graph pattern associated with Q to vertices over a corresponding match V_{T_i}.*

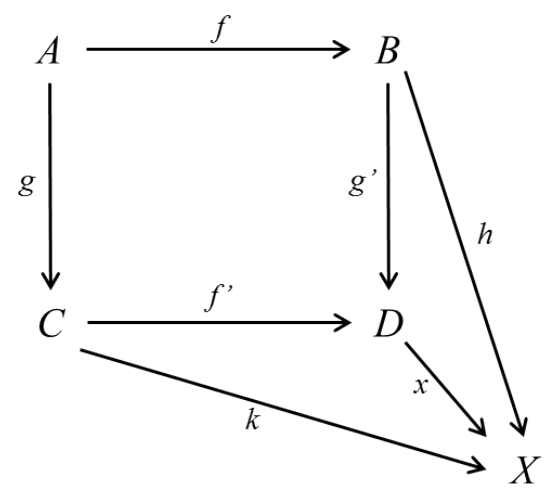
5.4 Inference Mechanism

So far, unified mechanisms for representing and retrieving information from heterogeneous models have been introduced. These mechanisms were based on the proposition that all formal models can, at some level of abstraction, be represented by a directed, labeled multi-graph. Such graph-based models were constructed through a purely syntactical transformation. However, no mechanism has been introduced to this point that allows for an *interpretation* of the graph-based models. In the following, such a mechanism is introduced for the purpose of performing *inference* in graph-based models.

In section 2.2, inference is introduced as the process of deriving logical conclusions from a set of *premises* by applying a series of *rules*. Several examples are given where the application of such inference rules lead to new statements being formed. For the case of graph-based structures, one can view such applications of inference rules as *transformations* of a graph through the process of adding (or possibly removing) vertices and edges. In the following, *graph transformation rules* are introduced as an enabling method for performing inference in graph-based structures. Graph transformation rules are also applied as a tool for semantic interpretation of the graph-based model, primarily by example.

5.4.1 Graph Transformations

The main idea behind graph transformations is the rule-based modification of graphs. Similar to the production rules first introduced in section 2.1, graph transformation rules (or *graph rewriting rules*) are defined with an antecedent and a consequent. For graph transformation rules, the *antecedent* (also known as the pattern graph) defines the context of the rule match, and the *consequent* (also known as the replacement graph) defines the modifications to be performed. In their typical definition, graph transformations enable two types of modifications: (1) adding and (2) deleting vertices



$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "g'"'] \arrow[ddr, "h"] & \\
C \arrow[r, "f'"] \arrow[drr, "k"'] & D \arrow[dr, "x"] & \\
 & & X
\end{tikzcd}$$

Figure 19: Illustration of the mathematical concept of a pushout (adapted from [184]).

and edges. A set of graph transformation rules is also called a *graph rewriting system* [184].

Formally, a graph transformation rule is a *production* $p : L \to R$, where L and R are complex patterns (see definition 5.4). L is known as the *left-hand side* and R as the right-hand side of the rule. Applying a rule p means finding a match to L in a source graph G and replacing the match to L by R, thus producing a *target graph* H. This transformation from a source graph G to a target graph H is typically depicted by $G \overset{p}{\Rightarrow} H$ [184].

In the related literature, a number of formal definitions for graph transformations exist. Commonly used in both model-based software and systems engineering is the algebraic approach based on *double-pushout constructions* [57]. A *pushout* is a mathematical concept from *category theory* [101] and can be defined in the following way: for two morphisms with common domains $f : A \to B$ and $g : A \to C$ the pushout over f and g is defined by a *pushout object* D and morphisms $f' : C \to D$ and $g' : B \to D$, for which $f' \circ g = g' \circ f$. Furthermore, the pushout must be universal, meaning that for any other pushout object X and morphisms $h : B \to X$

$$\begin{tikzcd}
L \arrow[d, swap, "m"] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Figure 20: Illustration of a double-pushout. For the double-pushout based definition of graph transformations, G is the source graph and H the target graph produced after replacing a match $m(L)$ to the complex graph pattern L with a replacement graph R. K is the gluing graph and is defined as the common subgraph of L and R. D is the context graph, defined through $(G \setminus m(L)) \cup m(K)$ (adapted from [184]).

and $k : C \to X$ for which $k \circ g = h \circ f$ there exists a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$. D is then typically referred to as the *gluing* of B and C. The definition of a pushout is illustrated in figure 19.

The *double-pushout* approach uses two gluing constructions to model a graph transformation. In this approach, the definition of a production is similar to before, with an additional graph K added: $p = (L, K, R)$. As before, L represents the pre-condition (i.e., the left-hand side to be matched) and R the post-condition (i.e., the right-hand side defining the replacement). K is defined as the *interface* between L and R (and of p) and can be thought of as the common subgraph of L and R. K is commonly referred to as the *gluing graph*. Therefore, the graph $L \setminus K$ describes the part to be deleted, and the graph $R \setminus K$ the part to be added when producing the target graph H. K is needed for context preservation when replacing the match to L with R.

In figure 20, a *double-pushout* diagram is shown. The double-pushout approach to graph transformations consists of two steps: first, construct a *context graph* D such that the gluing of L and D via K is equal to the source graph G. The context graph D is the source graph G minus the match to the left hand side L, unionized with the gluing graph K – i.e., $D = (G \setminus m(L)) \cup m(K)$, where $m(L)$ represents the match to the pattern L and $m(K)$ is the part of the match to L that is also shared with R. Thereafter, the gluing of R and D is constructed via K, leading to the target graph H. Note that if L is fully contained in R – i.e., $L \subseteq R$ – no vertices or edges are deleted when forming H[5].

A concrete example of a graph transformation rule applied to a data graph, and relations of the individual parts of the rule and data graph to definitions from the double-pushout approach are illustrated in figure 21. In the example, a graph transformation rule is utilized for adding a relationship *"installed_in"* as the inverse relationship of *"engine_type"* for any object that is known to be a particular kind of *Aircraft* (where, as done throughout this chapter, the relationship *"is_a"* signifies the *type-of* relationship).

Note that only an abbreviated version of the definition of the double-pushout approach to graph transformations is given. A more elaborate version can be found in [184]. It should be noted that a number of commonly applied graph transformation formalisms such as *triple graph grammars* (TGG) are closely related to this definition. Also, in practice, there exist a wide number of implementations of graph transformation frameworks (based on various graph models), many of which are used in Model-Based Systems Engineering research and applications (see, e.g., VIATRA [35], GReAT [8], booggie [100], FUJABA [154] and eMoflon [6]).

5.4.2 Inference using Graph Transformation Rules

As outlined in section 2.2, (deductive) inference is the process of opening new paths for inquiry by iteratively deriving statements through application of a set of rules to axioms and previously inferred statements. In the introduction to inference, examples

[5]Recall from chapter 2 that in the definition of production rules it is assumed that the left-hand side of the rule is preserved – this is a subtle, but important difference to the definition of graph transformation rules given here.

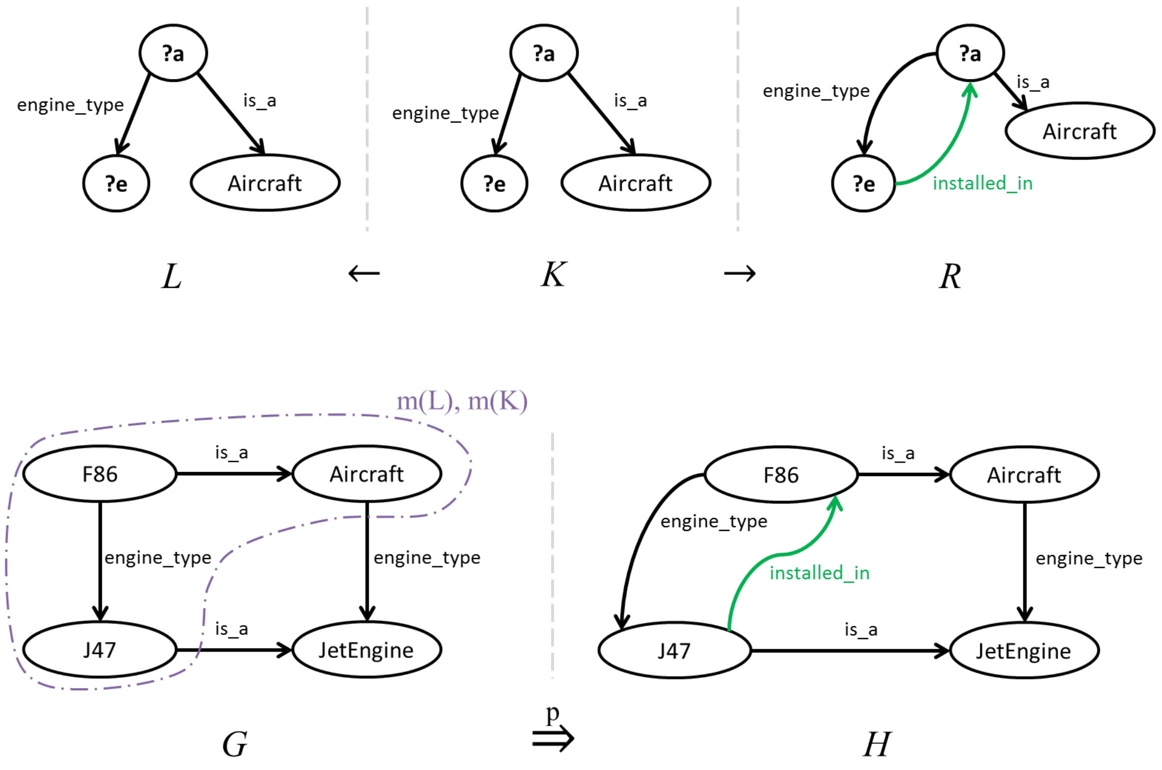


Figure 21: Graph transformation rule as an inference rule for inferring the inverse of the *"engine_type"* relation and its application to a graph G from which a graph H is derived.

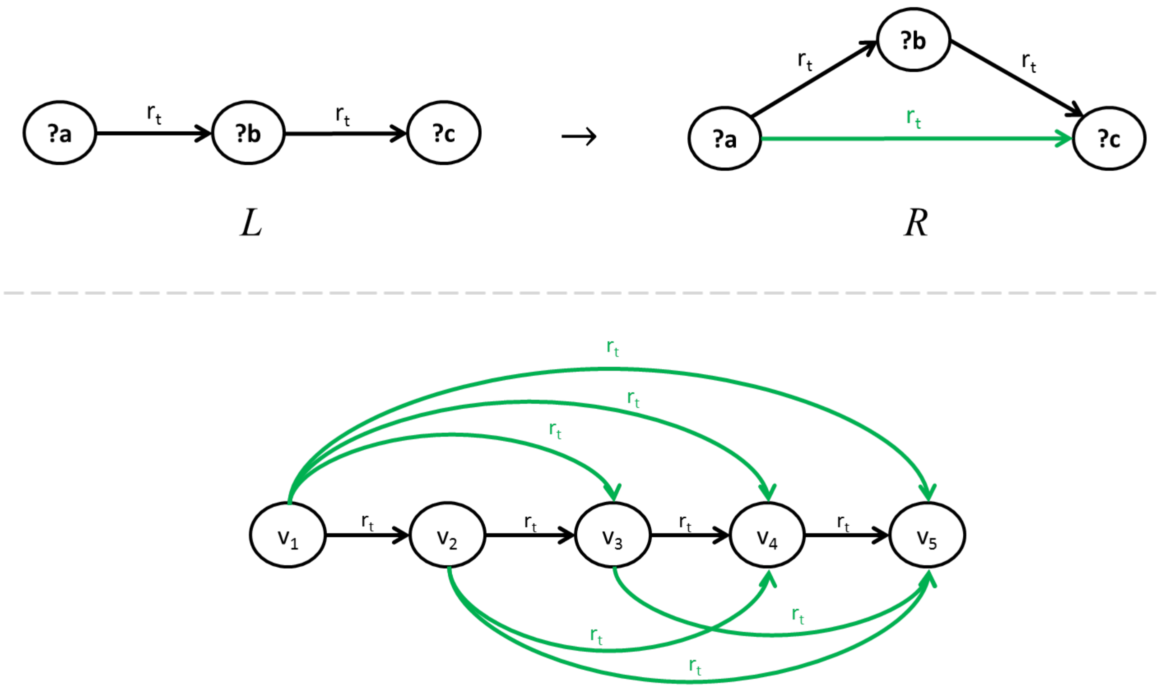


Figure 22: Graph transformation rule for inference of transitive relations (not showing the gluing graph K) (top) and a sample application to a data graph with inferred relations (bottom).

are given where implications written in a first-order language are applied. Here, the use of graph transformation rules for the same purpose are demonstrated.

Graph transformation rules, as defined in the previous section, can readily be applied for inference. Figure 22 depicts an example of a graph transformation rule for inferring relations as a result of the *transitive* nature of relation r_t and calculating the transitive hull of the relation r_t. Note that this is an example of the use of a graph transformation rule for interpreting language semantics – specifically, transitive constructs.

Graph transformation rules should be applied in combination with a data graph for identifying matches to a pattern without necessitating an exhaustive application of the graph transformations (i.e., the data graph should be considered as a representation of a set of *axioms and theorems* that are used in combination with a set of graph transformation rules which represent *inference rules*). This avoids having to expand a data graph by exhaustively applying all graph inference rules prior to extracting information from it (i.e., querying, or pattern matching) as would be done in a typical application of model transformations in MBSE. Similar to the discussion in section 2.2.3.1, such a mechanism can make use of forward- and/or backward-chaining, thereby also allowing for the inclusion of *recursive* inference rules (i.e., rules similar to those used for defining the sample context-free grammar in section 2.1.1.3), while still maintaining decidability for the membership problem (i.e., *is* $\phi \in \mathcal{L}$*?*). Otherwise, attempting to exhaustively apply a set of recursive graph transformation rules would lead to an infinite number of inferences and non-termination of the corresponding algorithm. Note that this is in alignment with the fact that only a very small set of languages is actually finite (see section 2.1.1).

5.5 Semantic Abstraction Mechanism

So far, mechanisms for translating information and knowledge encoded in heterogeneous formal models to a common, graph-based representation have been introduced. This introduction included a mechanism for performing basic inference in such graph-based models. However, since, per the presented mechanism, the formal models are transformed based on the terminology used in the corresponding language definitions, and semantic domains and mappings may not be transformed or unknown, little to no relations and interactions among the various models are known. This leads to a set of disconnected (or, in the best case, weakly connected) graph-based models. This problem is addressed by the introduction of a *semantic mediation* mechanism in the following.

The concept of semantic mediation described in this section is not a fundamentally novel concept. It has been inspired by concepts from the semantic web (see, e.g., [122]) and their application to tool integration [192]. However, its use for semantic abstraction and the formation of related semantic domains is novel.

5.5.1 Semantically Similar Concepts in Heterogeneous Models

While using different symbols, it can be argued that most languages share (at some level of abstraction) certain semantic concepts. In reference to the discussion in section 5.2, one commonality among formal modeling languages is the (abstract) concept of describing *objects* that have *properties*. Objects typically denote classes, subclasses or individuals of a class. Properties may be unary properties defining a state or quality of an object, or be defined as mappings.

To exemplify such commonalities, consider the concepts of a *SysML block* and a *Part* in the proprietary CAD tool Siemens NX. *Part*s are objects in the NX language representing physical components. *SysML block*s, on the other hand, *may* represent such physical components (depending on the modeling context and interpretation of the model) – however, in itself, the concept of a *SysML block* is far more abstract. Yet, both *Part* and *SysML block* share the common semantics that they represent a *predicable object* – i.e., something for which properties may be defined. For both *Part*s and *SysML block*s a common property is that of an identifier: i.e., a *name*.

The concepts of *object* and *property* are fundamental and are among the most general concepts known to mankind [135]. While seemingly abstract, the fact that both *Part*s and *SysML block*s can be identified as *objects* that carry certain *properties*, the semantics of which may be identical (such as the *identifier* of the object (i.e., its *name*)), lets one express basic (yet still semantically abstract) relationships among formal models. This enables the *differentiation* of some terms – even if just to a limited extent – and the identification of membership of a particular class of things. This is useful for reasoning at a higher level of abstraction. In the following, a set of common terms (such as *object* and *property*) that syntactic expressions are mediated to is referred to as a *mediation vocabulary*.

5.5.2 Mediating Expressions

Within the context of this dissertation, semantic mediation is defined as the process of relating elements of models that have a specific semantic meaning to a concept that has an equivalent, or more abstract (or general) meaning. Semantic mediation takes into account a-priori definable similarities among languages based on their semantic interpretation. Mediation acts as a translation and semantic abstraction mechanism. An entity mediating expressions is defined as a *mediator*.

In alignment with the framework presented in this chapter, concepts from the mediation vocabulary are represented as graph structures. Mediation occurs through the application of a graph inference rule (see section 5.4). This graph inference rule adds a relationship (in the form of a graph edge) to a particular syntactic element, thereby representing the explicit mapping of the syntactic element to an concept from the mediation vocabulary.

5.5.3 Base Vocabulary

As part of the research, a semantic mediation vocabulary defining concepts that are shared among commonly used engineering models was empirically derived. The basis for this was formed by analyzing common features of class diagrams and object-oriented models. In addition, the related literature from the domain of language theory was analyzed for classifications of semantic relationships common to all languages (of which part-whole, predecessor-successor, instance of, and synonym are four examples [207, 86]). Furthermore, theories related to design and systems engineering, such as the *Rational Design Theory* [216] have served as a source for inspiration. Lastly, language definitions of well-known calculi, such as various flavors of description logic [7], were analyzed (particularly due to their use in information and knowledge capturing), as well as widely accepted modeling languages such as UML. Initial findings of the investigation are reported in [173] and, partially, in [67].

It should be noted that the empirical evaluation and continued evolution of the vocabulary (and aspects of the mediation mechanism in general) were conducted as part of a joint research effort between the Institute of Automation and Information Systems, Institute of Product Development, and Chair for Information Systems at the *Technische Universität München* (TUM), and the *Model-Based Systems Engineering Center* (MBSEC) at the Georgia Institute of Technology. Initial results are published in [67][6]. With the evaluation being based only on an empirical investigation consisting of the application of the mediation vocabulary to a single system (a *pick-and-place unit* [225]) for the purpose of performing analyses across heterogeneous models, no claim for completeness of the vocabulary is made (however, it is applied in a case study in chapter 8 where a different system is under study).

5.5.3.1 Base Concepts

In MBSE, the process of specification and analysis is supported through the application of formal modeling [72]. As described in section 5.5.1, two fundamental semantic concepts encountered in any formal modeling language are that of an *object* and that of a *property*. In the base vocabulary, these concepts are denoted *Entity* and *Relationship*, which are marked explicitly as specializations of the most general term of *Base Concept*s. This is illustrated in figure 23.

The rationale behind choosing *Entity* rather than the previously used term *Object* is that, in object-oriented design and model-based engineering, the term *Object* is typically understood to be "an entity that has state, behavior, and identity" [22]. *Object* is related to the term *Class* in that "the structure and behavior of similar objects are defined in their common class" [22]. Here, the term *Entity* is intended to

[6]Note that the full extent of the results of this joint work are unpublished as of the date of writing this dissertation. Aside from the author of this dissertation, the researchers involved are Stefan Feldmann, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Dr. Ahsan Qamar, Prof. Dr. Christiaan Paredis, Prof. Dr. Birgit Vogel-Heuser, Prof. Dr. Helmut Krcmar, Prof. Dr. Lindemann

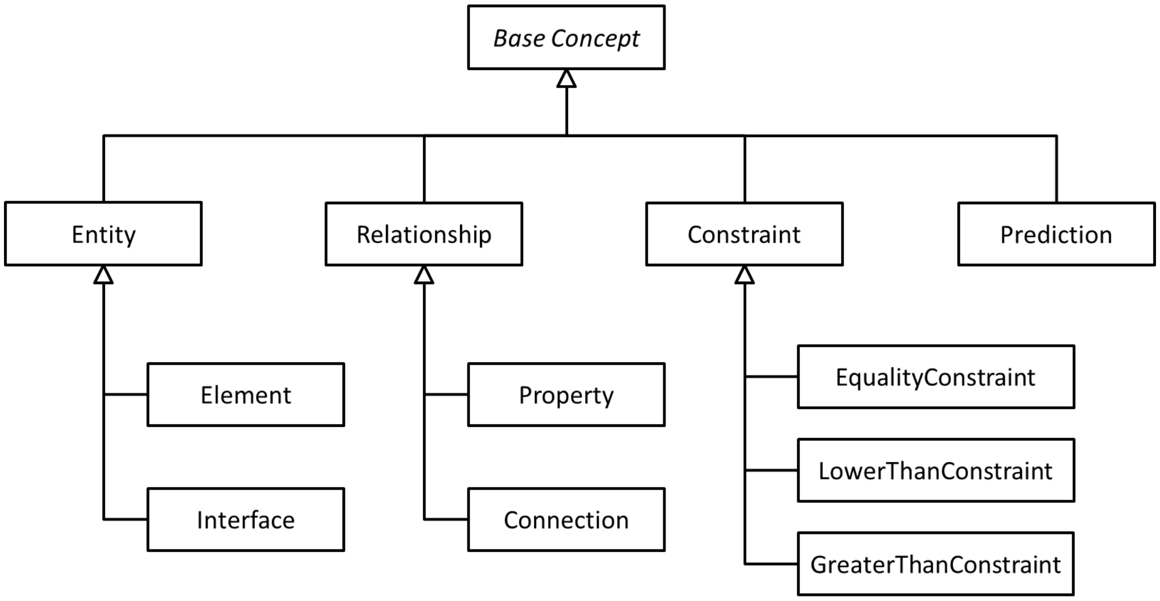


Figure 23: Inheritance hierarchy of base concepts in the base vocabulary in standard class diagram notation.

be an overarching concept for *Object* and *Class*, thereby removing the relative notion of the context-specific *"instance of"* relationship between *Object*s and *Class*es (which, recalling from section 2.1, can be thought of in relative terms in defining modeling languages using multiple levels). The base concept *Relationship* is used in denoting the general concept of something that represents a kind of *property* of an object, which can be thought of as a relation to either itself or between two or more entities.

Fundamental to designing (engineering) systems are the processes of *specification* and *analysis*. Specification involves *constraining* an initially infinite set of *alternatives* by restricting the ranges of *properties* of a system to be designed (by imposing constraints), where properties may be of a structural or numeric nature. Within the context of engineering, analysis involves the careful study of, and prediction of qualities and properties of a system that are (potentially) subject to uncontrollable (or unaccountable) environmental influences, unforeseen phenomena, or simply randomness (e.g., variance of mass and length dimensions of a machined part, cost of a system over its lifecycle, and demand for a product). Predictions of outcomes of a real world process are inherently uncertain *rationalDesignTheory*. These concepts are made explicit with the base concepts *Constraint* and *Prediction*, which are to be understood to be *imposed over* properties.

The concept of an *Entity* is further specialized into two subclasses: *Element* and *Interface*. *Element*s are distinguished from *Interfaces* mainly for practical purposes and to enable a clear distinction between abstract definitions from their concrete implementations, thereby also allowing for the concept of references. Clearly, an *Interface* is related to one or more *Element*s, but is not a concrete implementation of such. Primitive types typically encountered in modeling languages, such as string literals, are interpreted to be *Element*s. *Relationship*s are further refined into *Property*s and *Connection*s. *Property*s are relationships that can be unary in nature, and are understood to be an integral part of the definition of a particular concept. Connections, on the other hand are not part of the definition of a concept and merely relate two concepts to one another – e.g., through an interface. *Constraint*s are further specialized into the mathematical concepts of imposing a single value (*EqualityConstraint*) or a range of values (*LowerThanConstraint* and *GreaterThanConstraint*). The concept of a *Prediction* is not specialized further.

Note that the base terms imply a slightly broader spectrum of concepts than that typically utilized by model-based (and model-driven) software engineering approaches. These approaches are typically based on incorporating concepts from *mathematical logic*, whereas the concepts outlined in this section refer to concepts from a broader subset of mathematics – for instance, the concept of *predictions* is from the mathematical domain of *probability theory*. This is necessary, since the base vocabulary is meant as a basis for developing models of physical systems, select properties of which are inherently uncertain.

Table 1: Overview of standard properties defined in the base vocabulary (instances of the base concept *Property*).

Property	**Domain**	**Range**	**Description**
type	BaseConcept	BaseConcept	Instance relation; relation between a general concept and individual instances.
contains	BaseConcept	BaseConcept	(Weak) containment relation (e.g., part-whole or object-property).
containedIn	BaseConcept	BaseConcept	Inverse of above.
generalizationOf	BaseConcept	BaseConcept	Hyponymous relation; denotes (sub-)class membership.
specializationOf	BaseConcept	BaseConcept	Inverse of above.
domain	Relation	BaseConcept	The source domain of a relation.
range	Relation	BaseConcept	The target domain of a relation.
equivalentTo	BaseConcept	BaseConcept	Semantic equivalence of base concept; synonymy; identifies that both source and target elements have the same meaning.
differentFrom	BaseConcept	BaseConcept	Inverse of above.

5.5.3.2 Standard Properties

Table 1 lists a number of concrete properties (i.e., instances of *Property*) that various concepts may possess (indicated by the domain). Note that the set of properties and concepts is meant to be a self-referential set, enabling the bootstrapping of the vocabulary similar to the idea of bootstrapped (meta-)models (see section 2.1). For instance, the property *type*, denoting an instance-of relationship, is defined by itself as an instance of a *Property.* The addition of domain and range in the table constrain the well-formedness of base expressions and must be adhered to when mediating to expressions from the base vocabulary.

Table 2: Overview of standard properties in the base vocabulary introduced for conveniently expressing common concepts.

Property	**Domain**	**Range**	**Description**
name	BaseConcept	Element	Identifier object.
value	Constraint	Element	Value associated with a constraint.
unitType	Constraint	Entity	Unit type associated with a constraint value.
constrainedBy	BaseConcept	Constraint	An applied constraint.

The list of concrete properties accounts for part-whole relations, instance-of relations, hyponymous relations (generalization / specialization) and synonymous relations. Note that relations such as those denoting the predecessor-successor relationship are not included and are assumed outside the realm of the base vocabulary (since these are specific to certain types of models, rather than being common to all, or common to relations across models). Not included are also common relationships from the related language literature denoting the concepts of a *homonym* and *antonym*, since the applications and uses of the base vocabulary have not shown a significant need for these.

Table 2 lists an additional set of properties, which are introduced primarily for convenience and due to their wide use in modeling languages. This includes an attribute denoting the *name* of a concept (or its instance), which acts as an identifier. Furthermore, concepts related to *Constraint*s are included. Note that, fundamentally, the concept of a *Constraint* and that of an *attribute* (which, here, is modeled as a *Property*) are closely related, in that both are mappings to a range (where, for *Constraint*s the range is defined by its *value*, and for non-unary *Property*s, the value is assigned explicitly through a constraint). However, in the empirical evaluation, the differentiation between the concepts has shown to be practical. Additionally, the concept of a *unitType* property has been introduced, allowing for the explicit expression of unitized values. This is similar to how OMG SysML treats units [72, 160]. In the initial exploration, this has shown to be more convenient and pragmatic than expressing unitized types as specializations of numeric base types such as is done in the *Modelica* language [74] (hence eliminating the need for separate unit types).

Note from table 2 that not only *Property*s can be constrained, but any *Base Concept*. The reason for this is to allow for the representation of constraints on concepts such as *Relation*s – e.g., for the purpose of indicating cardinality constraints. The use of *Constraint*s for this purpose has not yet been explored, but is planned in future research.

5.5.3.3 Mediation Example

Figure 24 illustrates an example mediation of a graph-based model to the base vocabulary. The example model used is an illustrative part of the model used in demonstrating the translation to the graph-based representation as depicted in figure 15. Note that the meta-meta model elements are not shown. Note in figure 24 (b) how the mediation of the property *members* results in the construction of two additional nodes (for which the label is irrelevant), one of which is a concrete instance of the property *members*, the other being a concrete *Constraint* over the property.

Note that for reasons of brevity and to aid readability, not all mediations are shown. A universally valid set of mediation rules cannot be constructed, since each set of mediation rules is specific to a particular modeling language. Such transformations must necessarily be created by an entity that is capable of interpreting the language to be mediated (in addition to the base vocabulary terms). Hence, it is assumed that a human defines a set of mediation rules (as graph inference rules) for each involved modeling language.

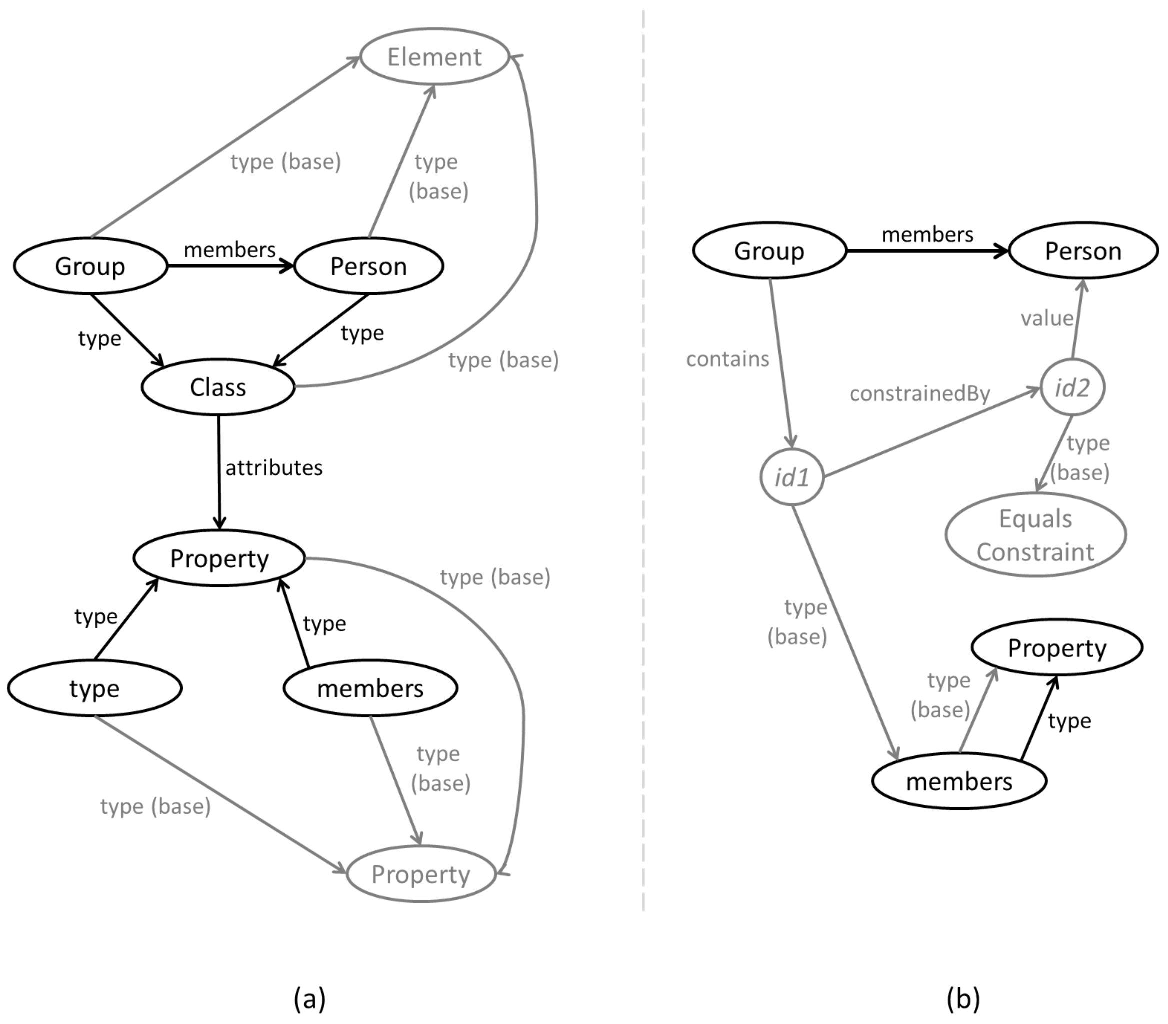


Figure 24: Mediation of an illustrative part of the model from figure 15. Gray nodes and edges represent elements from the base vocabulary. Note that such mappings are language-specific and must be defined by an external entity based on the understanding of the language to be mediated. Here, only an illustrative subset of the mediations is shown: (a) shows the mediation of model elements of type *Class* to *Element*, where, by definition of an externally defined mapping, all instances of a *Class* are also *Elements* (transitivity of *type*); (b) demonstrates the mediation of properties, which are assigned to the predicated objects through a containment relation (properties define objects and are *part of* the definition of an object). Note that all *type* (denoting "instance of") relations native to the language definition of the translated model are mediated to the *type* concept from the base vocabulary.

It should also be noted that a mediation *must not necessarily be complete*[7]. The key motivating factor behind mediation is, as outlined previously, to aid in making information available to mechanisms that perform reasoning at higher levels of semantic abstraction. However, no assumption is made about the completeness of this information, which is in line with the concept of abstraction. Only very few well-formedness constraints are imposed.

5.5.4 Domain- & Language-Specific Vocabularies

While the concepts introduced in the previous section are common to most (if not all) heterogeneous models and their respective modeling language definitions, there are some sets of related concepts that are only common to defined *subsets* of models and modeling languages. Within the context of this dissertation, mediation vocabularies that contain such concepts are referred to as *domain vocabularies*. It is assumed that no closed set of such domain vocabularies can be elicited. Rather, it is assumed that a (possibly infinite) number of partially, or fully overlapping domain vocabularies exist.

Domain vocabularies are similar to a base vocabulary for a specific language, application, or domain. Examples of domain vocabularies include a vocabulary for the domain of *mechatronics*, or for the domain of *requirements*. A vocabulary for the domain of requirements may include concepts such as a *Requirement*, which maps to the semantically less precise concept of an *Element* in the base vocabulary. Two or more vocabularies may also overlap: for instance, a vocabulary for the domain of *systems engineering* may overlap partially with both of these (either syntactically, or by definition of mediation rules between the respective vocabularies).

Note that per this definition of various sets of vocabularies, the vocabularies used in translating formal models from their corresponding serializations or representations to a graph-based representation can be thought of as *language-specific vocabularies*.

[7]This is one of the reasons why the introduction of a *meta-model* for the base vocabulary or a representation as an ontology was avoided.

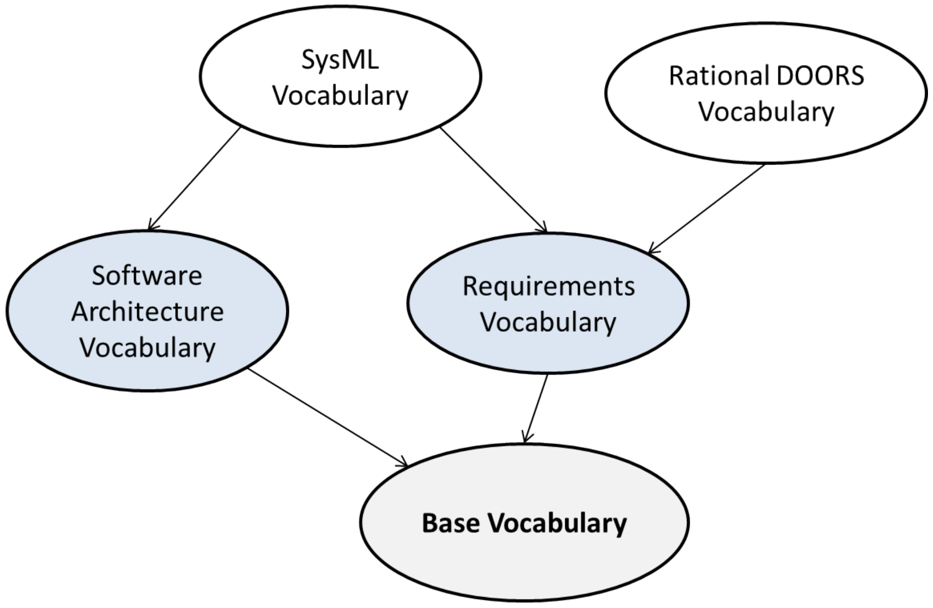


Figure 25: Illustration of the concept of multiple semantically overlapping mediation vocabularies. Note that Rational DOORS is a requirements management tool, and SysML a general purpose modeling language. Arrows denote sets of semantic mediations.

These language-specific vocabularies may include terms relevant to one or more known domain vocabularies: for instance, a subset of the language vocabulary for *SysML* overlaps with the concepts described in a domain-specific vocabulary for *requirements*.

Introducing multiple, related vocabularies allows for multiple levels of semantic abstraction to be defined at the same time. Which level of semantic abstraction is being referred to depends on which terms from the vocabulary are being referred to. This leads to a variance in semantic precision starting from the base vocabulary over various domain vocabularies to language-specific vocabularies. This is illustrated in figure 25.

Note that the concept of multiple vocabularies also increases reusability, and positively supports maintainability and extensibility. Once a mediation from a domain-specific vocabulary to the base vocabulary has been defined, the mediation from a language-specific vocabulary to the base vocabulary is *implicitly defined* once a mediation to a domain-specific vocabulary has been defined.

5.5.5 Unit Mediation

The concept of mediation is not limited to a static mapping to syntactic constructs (representing semantic concepts) from a vocabulary. In some cases, it makes sense to define mediation rules that allow for dynamic conversions to be performed by inference. For instance, in the performed research, a (partial) domain vocabulary for *physical units* was constructed (based on [111]). This vocabulary defines the concepts of units, quantity kinds and dimensions. The developed vocabulary also defines *feet*, *meters* and *kilometers*, where *meters* is marked as a *base unit*. Similar to before, mediation rules from language-specific definitions of physical units to expressions from this vocabulary of physical units are defined. All physical units have an attribute denoting a *conversion factor*. Whenever a value with a defined unit is encountered, and the unit is not a base unit, an equivalent value corresponding to a base unit type can be *inferred* through a defined graph transformation rule that uses the conversion factor.

5.6 Summary

In this chapter, a formal and sound basis for representing, retrieving information from (querying), and manipulating (transforming) information and knowledge encoded in heterogeneous models is presented. The first part, comprised of sections 5.1 and 5.2, discusses the (syntactical, propositional) representation of the information and knowledge encoded in formal models, and how syntax and semantics can be transformed to a unifying, graph-based formalism. The second part, comprised of sections 5.3 and 5.4 discuss the retrieval of information and manipulation (for purposes of transformation and inference) of models that are represented by graphs. The last part of the chapter introduces a semantic abstraction mechanism, which allows for higher level reasoning.

The common representational formalism is built on concepts from graph theory and graph transformations. Directed, labeled multi-graphs are utilized for representing the information and knowledge encoded in models in *propositional* form. Graphs are a practical model for this: in general, atomic propositions can be represented by a *subject-predicate-object* triplet. If the subject and object are depicted by vertices, and the predicate is a directed edge between the vertices, the proposition is represented by a graph. Pattern matching and graph queries are formally introduced as means to retrieve information from such graph-based representations of models. Graph transformations are then formally introduced as a basis for performing inference in, and generally transforming graph-based models. The use of graph transformations and graph structures for defining formal semantics is discussed as well. Using the developed concepts as a basis, a formal system for heterogeneous models can be constructed.

Finally, a semantic mediation mechanism is introduced. The process of mediation and, in particular the introduced *base vocabulary*, enables heterogeneous models to interface and interact. A key concept and property of the presented approach is the exploitation of language-specific concepts to infer semantic information for higher-level reasoning applications (e.g., reasoning over all entities in models considered *objects* rather than just all *SysML block*s). To some degree, the concept of semantic mediation is similar to that of tagging models with elements from one or more ontologies (see section 3.3.2). However, it is different in the sense that it is a largely automated process, where the (semantic) mapping from syntactic expressions to elements of the ontology (which can be understood to be a representation of the semantic domain (see section 2.1.1.4)) is defined by graph inference rules.

CHAPTER VI

PROBABILISTIC INEXACT REASONING OVER GRAPH-BASED MODELS

In this chapter, a generic approach to inexact probabilistic reasoning over graph-based models allowing the abductive inference of propositions with uncertain truth values is presented. The approach is based on a combination of pattern matching in graphs, logical inference and Bayesian inference. Therefore, it builds on the concepts introduced in chapter 5, and is a concrete underlying method for implementing the inconsistency identification framework from section 4.3. As briefly discussed in chapter 3, approaches to reasoning under uncertainty in ontology and database models have been reported in the related literature, but only very few are sound (and Bayesian), and none address the case of reasoning over heterogeneous models.

The goal of this chapter is to address, in part, research question 3. In doing so, a generic approach to applying Bayesian inference to reasoning over graph based models (see hypothesis 4) is presented, and its technical feasibility and viability demonstrated by the introduction of an algorithm. In previous chapters, the necessary background and basis for the developed concepts herein are already introduced. Section 2.3 introduces the necessary background in probability theory and Bayesian reasoning, and chapter 5 develops an approach for representing, retrieving, modifying and logically inferring statements from information and knowledge encoded in heterogeneous models.

The chapter is outlined as follows: first, a brief review of inexact reasoning methods, and a rationale for choosing a Bayesian approach is presented. Thereafter, an overview of the proposed approach is introduced in detail. Part of this introduction is also a discussion on what a probabilistic inexact reasoning model is, and how it can be set up in practice. In addition, aspects of how the argument chain can be stored in a graph-based fashion is discussed. The approach is demonstrated using a simple example from the literature. Towards the end of the chapter, algorithmic procedures for implementing the approach are presented (section 6.3.1). The chapter closes with a brief summary of key aspects of the approach and the most important insights.

6.1 Inexact Reasoning

A formal deductive apparatus assumes a set of *factual* statements (i.e., the axioms) and inference rules as its basis for deriving statements and reaching conclusions (see section 2.2). Derived statements are logically valid conclusions (entailments) based on the relationships between true statements and those that (logically) follow from these statements. As mentioned in section 2.2.3, such a deductive apparatus is the basis for forming a theory. Given a definition of the formal semantics, each well-formed formula and each relationship between true statements have an unambiguous meaning. Reasoning under these assumptions is introduced in section 2.2.3.2 as *exact reasoning*.

Inexact reasoning is reasoning performed using inductive or abductive inference, and covers reasoning tasks in situations where there is any one, or a combination, of the following conditions present [81]:

- Inexact or vague statements and rules *(i.e., the truth or well-formedness of some statement ϕ is not clear, but a belief about its truth is quantifiable)*
- Incomplete statements and rules
- Contradictory statements or rules *(i.e., somehow ϕ and $\neg\phi$ are both considered sufficiently likely to be true)*

In inexact reasoning the antecedent, the conclusion and even the meaning of an inference rule can be uncertain [81]. An inexact reasoning mechanism for formal models is meaningful, since models of systems with physical properties are inherently abstract and incomplete. When evolved concurrently, incorrectnesses may also be introduced. Furthermore, assumptions flow into models, and the relations among models are not known explicitly. These assumptions, in addition to the incompleteness of models, results in the *interpretation* of models no longer being unambiguous. This leads to ambiguity in the definition of the underlying formal system resulting from composing the various models describing a system, as well as complex, implicit relations among models that are not known with certainty.

In the following, three predominant approaches to inexact reasoning from the literature that are based on probability theory are briefly introduced and compared. The goal of this section is to offer a rationale for selecting Bayesian probability as the basis for an inexact reasoning method. This is primarily done by making issues in the formality of the approaches explicit.

6.1.1 Approaches to Probabilistic Inexact Reasoning

In the related literature, numerous approaches to inexact reasoning over *ambiguous databases* – that is, databases of statements with uncertain truth values – are reported (e.g., [215, 65, 186, 17]). However, while acceptedly the most sound [81], very few of the approaches used in practice are based on Bayesian probability theory. Well-studied and commonly applied approaches to probabilistic inexact reasoning include *certainty factor* (CF) theory [194, 1] and *Dempster-Shafer theory* [190]. In the following, these approaches are introduced and the rationale for their use in practice is discussed and compared.

6.1.1.1 Bayesian Probability Theory

Fundamentals of Bayesian probability theory and Bayesian reasoning have been introduced in section 2.3. Within this framework, the cornerstones for inexact reasoning are the definition of conditional probability (see equation 2) and, in particular, the application of Bayes' theorem (see equation 13). Bayes' theorem is used for *updating* a *prior belief* with observed evidence to form a *posterior belief*. In machine learning applications, Bayesian inference is commonly applied to *diagnostic reasoning* (particularly in the medical field [81]). Given a model of a probabilistic experiment, diagnostic reasoning is the process of evaluating the truth of a possible (but unobserved) event, given a prior belief about this event and a set of related events. For instance, a common example from the medical domain is determining whether or not a particular patient has cancer. The probability of a random patient having cancer (given no observations about the patient, other than the acknowledgement of his or her existence) is, acceptedly, very low. However, after observing a patient, and gathering evidence in support or opposition of the *hypothesis* that the patient in question has cancer, this prior belief is updated. For instance, the observation of whether or not the patient has a history of smoking is information that updates the prior belief about the patient having cancer. This updating can be done by applying Bayes' theorem.

Bayesian inference is widely regarded to be the most formal, and (mathematically) sound approach to inexact reasoning. Therefore, it is no surprise that it is also one of the earliest concepts used in artificial intelligence and machine learning. However, in machine learning practice, the fact that a large number of (consistent) values are required for defining the probabilities needed in applying Bayes' theorem is often considered to hinder its practicality. Typically, Bayesian inference is only applied if large datasets are available, from which frequencies can be extracted that are considered sufficiently accurate representations of probabilities for future events.

Bayesian inference is also used if assumptions about the independence of events can be made (see the motivation behind Bayesian networks described in section 2.3.3).

6.1.1.2 Certainty Factor Theory

Certainty factor theory was originally developed for the medical diagnosis expert system `MYCIN`. CF theory was developed on the premise that a consistent set of probabilities is "difficult if not impossible [sic]" to elicit [81]. The difficulty in identifying a *consistent* set of probabilities was discovered by the developers of `MYCIN` when asking medical experts to judge the accuracy of inferred beliefs. For instance, in [81] the following example is mentioned: *"say that (1) the stain of the organism is gram positive and (2) the morphology of the organism is coccus and (3) the growth conformation of the organism is chains then (c) the identity of the organism is streptococcus with probability 0.7"*. It was found that, while the medical experts agreed that this conclusion is correct, they were not ready to accept the probability of the mutually exclusive event. That is, medical experts did not agree that *"the identity of the organism is* ***not*** *streptococcus"* with probability 0.3 is consistent with their beliefs.

The conclusion drawn by Shortliffe and Buchanan is that, while a set of observations E influence the probability of an event H, the same set of observations may not influence the complementary event $\neg H$ [195]. Based on Carnap's theory of confirmation [28], Shortliffe introduced the concept of certainty factors as a *degree of confirmation*, which he defined as the difference between belief and disbelief[1]:

$$\mathrm{CF}(H, E) = \mathrm{MB}(H, E) - \mathrm{MD}(H, E)$$

In the above equation, MB is the *measure of increased belief* in H due to E and MD is the *measure of increased disbelief* in H due to E. These measures are defined in

[1]The definition was altered slightly in later work, where the term $\mathrm{MB}(H, E) - \mathrm{MD}(H, E)$ is divided by $1 - \min(\mathrm{MB}, \mathrm{MD})$ to "soften the effect of a single piece of disconfirming evidence on many confirming pieces [sic] of evidence" [81].

the following way:

$$\mathrm{MB}(H,E) = \frac{\max\left[P(H \mid E), P(H)\right] - P(H)}{\max[1,0] - P(H)}$$

$$\mathrm{MD}(H,E) = \frac{\min\left[P(H \mid E), P(H)\right] - P(H)}{\min[1,0] - P(H)}$$

Note that for the case of $P(H) = 1$, CF theory states that $\mathrm{MB}(H,E) = 1$. Similarly, for $P(H) = 0$, CF theory states that $\mathrm{MD}(H,E) = 0$. Certainty factors are defined in the interval $-1 \leq \mathrm{CF}(H,E) \leq 1$, where $\mathrm{CF}(H,E)$ is the case of "no evidence". The event with the higher certainty factor is then considered `true`.

It is important to understand that CF theory was developed as a basis for an *ad hoc* method to determine the applicability of a deductive *rule* [81]. In `MYCIN`, for instance, any time the value of the certainty factor is determined to be $\mathrm{CF}(H,E) > 0.2$, the antecedent of a rule is considered `true`, thereby deducing the truth of the consequent. While `MYCIN` was, reportedly, successfully applied in diagnosing diseases, it is an ad hoc method that lacks an underlying formal theory, and it is impossible to guarantee that the use of CF theory produces valuable results for similar applications. A major formal flaw is the fact that, while CF theory is based (partially) on probability theory, it violates one of the basic axioms of probability theory (see Kolmogorov's axioms in section 2.3.1.1). This can lead to non-sensical deductions, such as that it is possible for a disease H_1 to have a higher conditional probability $P(H_1 \mid E)$ than a disease H_2 (i.e., $P(H_1 \mid E) > P(H_2 \mid E)$) given the same observations, but, at the same time also a lower certainty factor (i.e., $\mathrm{CF}(H_1,E) < \mathrm{CF}(H_2,E)$)[2].

6.1.1.3 Dempster-Shafer Theory

Dempster-Shafer theory (also known as *theory of belief functions* and *evidence theory*) is based on the idea of modeling uncertainty by a range of probabilities rather than

[2]A concrete example of this can be derived by computing the certainty factor using the assumed values $P(H_1) = 0.8$, $P(H_2) = 0.2$ and $P(H_1 \mid E) = 0.9$, $P(H_2 \mid E) = 0.8$.

a single probabilistic number [41, 190]. Dempster-Shafer theory is generally accepted to have a "good [sic] theoretical foundation" [81].

A fundamental concept in DS theory is the notion of an *environment* and *frame of discernment*. Environments are (finite) sets $\boldsymbol{\theta} = \theta_1, \theta_2, ...$, elements (or subsets) of which are interpreted as "possible answers to a question" [190]. Elements of such an environment are considered mutually exclusive and the set itself is assumed exhaustive. Note that the null-set $\emptyset$ is not considered a valid answer by assumption of the exhaustiveness of the environment. An environment is called a frame of discernment if its elements are possible answers to a question, but only one answer is correct.

Fundamentally different between the concept of (Bayesian) probability theory and Dempster-Shafer theory is the treatment of *ignorance*. For instance, if no prior knowledge exists, the *principle of indifference* states that the probability of each event is equally likely. Even if a probability is assigned to only one event A out of two possible events A and $\neg A$, the laws of probability theory "force" [sic] the assignment of a probability to $\neg A$ through $P(\neg A) = 1 - P(A)$ "even if there is no evidence for this" [sic] [81]. Dempster-Shafer theory does not force a belief to be assigned: instead, probabilities are assigned to only those subsets of the environment to which one wishes to assign a belief. Any belief not assigned to a specific subset is considered a *nonbelief* associated with the environment. For instance, assume an environment $\boldsymbol{\theta} = \{\theta_1, \theta_2, \theta_3, \theta_4\}$. A belief is now elicited on the subset θ_1, θ_3. Say this belief is $m(\{\theta_1, \theta_3\}) = 0.6$; the nonbelief associated with the environment is then $m(\boldsymbol{\theta}) = 1 - 0.6 = 0.4$. Note that m is known as the *mass function* which assigns a number in the interval $[0, 1]$ to any element of the power set of $\boldsymbol{\theta}$. The sum of all masses must equal 1 (note the similarity to the *probability measure* from section 2.3.1.1). Nonbelief is interpreted as "neither belief nor disbelief in the evidence to a degree of 0.4" [81] – i.e., 0.4 is *not* specifically assigned to any subset of $\boldsymbol{\theta}$. This is also where the difference of the definition of the mass function and that of the probability measure becomes evident (i.e., if $\boldsymbol{\theta}$ were the set of possible *outcomes* of an experiment, the probability of $\boldsymbol{\theta}$ is, by definition, 1).

Evidence is combined in DS theory using Dempster's *rule of combination.* One form of this rule is:

$$m_3(Z) = m_1 \oplus m_2(Z) = \sum_{X \cap Y = Z} m_1(X) m_2(Y)$$

The formed mass m_3 is a *consensus* of the original evidence and "tends to favor agreement rather than disagreement" [81]. An important characteristic of the rule is that it is used to combine evidence that has independent errors, which is not meant to be understood as independently gathered evidence. Note that in some cases, the sum of the masses defined by m_3 may not equal 1 after combining them. In such cases, the results are *normalized* to 1.

The *contribution* of a belief in an event is made explicit using the concept of an *evidential interval* EV. Evidential intervals are defined by a *minimum* and a *maximum belief* in an event. Such *belief measures* are computed as the mass assigned to a set and all its subsets (for cases when no mass is assigned to a subset, the mass is simply 0). For instance, the belief $\text{Bel}(\{\theta_1, \theta_2\}) = m(\{\theta_1\}) + m(\{\theta_2\}) + m(\{\theta_1, \theta_2\})$. The evidential interval is then defined as $\text{EI}(\{\theta_1, \theta_2\}) = [\text{Bel}(\{\theta_1, \theta_2\}), 1 - \text{Bel}(\boldsymbol{\theta} \setminus \{\theta_1, \theta_2\})]$. An evidential interval $[1, 1]$ may be interpreted as *completely true*, and $[0, 1]$ as *indifferent.*

DS theory is typically used for data fusion (e.g., when data from sensors is collected), and whenever this data is uncertain or imprecise data (see, e.g., [182] for an application). While attractive from the perspective that the assignment of masses does not have to be complete – i.e., it can be done over an arbitrary subset of events – Dempster-Shafer theory has a fundamental flaw: as mentioned earlier, there are cases in which the sum of the masses does not equal 1. Normalization is one possibility of ensuring that this criteria is fulfilled. However, it is exactly this normalization of beliefs that can lead to unintuitive, and unexpected results: Zadeh shows this in [233] using an example from diagnostic reasoning in medicine [81]. Say there are two doctors, A and B, each of which has beliefs on the patient's illness. The beliefs of doctor A may be expressed as follows: $m_A(meningitis) = 0.99$, $m_A(braintumor) = 0.01$. Doctor B's beliefs take the form of $m_B(concussion) = 0.99$ and $m_B(braintumor) = 0.01$. Therefore, $\theta_{\mathbf{A}} = \{meningitis, braintumor\}$ and $\theta_{\mathbf{B}} = \{concussion, braintumor\}$. Combining these beliefs leads to $m_{AB}(braintumor) = 0.0001$. All other values of m_3 are 0. However, since the sum of all values assigned by m_3 must equal 1, it follows, after normalization, that $m_{AB}(braintumor) = 1$, which is unintuitive and non-sensical, since both doctors agree that the probability of a brain tumor is very low.

6.1.2 Comparison of Approaches

As is evident from the identified limitations, Bayesian probability theory is the only mathematically sound and formally correct framework for inexact reasoning among the three introduced approaches. CF theory is generally acknowledged to be ad hoc [81] and Dempster-Shafer theory – while claimed to be formal and sound – can, under specific circumstances, produce unintuitive results. Additionally, the degree of formality is often based on the claim that Dempster-Shafter theory is a *generalization* of Bayesian probability theory [42]. However, it can be shown that this assumption actually leads to an inconsistency in Dempster-Shafer theory [229]. While DS theory has been used in research related to information fusion and artificial intelligence with success, evidence exists that its validity and soundness is not a given. This is partially due to the aforementioned reasons. However, it is also due to its non-applicability to *general* evidence combination, and only to certain types of scenarios (which are not clearly identified) [43].

CF theory is motivated by the fact that, given the same observations, inferred probabilities are not always intuitive to a human. However, it can be argued that this is either a result of (1) an incomplete definition of the underlying experiment or

(2) an inappropriate belief elicitation process. Even if (1) were the case, (2) may still be problematic. In support of (2), the accepted method for eliciting subjective beliefs is that originally devised by de Finetti [40], where probabilities are not interpreted as frequencies, but as a *willingness to bet* on the occurrence of a phenomenon: i.e., as a rate at which an individual is willing to bet on the occurrence of an event [187, 178]. Recall that the method used for eliciting knowledge in `MYCIN` did not make use of this.

The use of reasoning methods that lack a sound mathematical foundation is problematic due to their ad hoc nature and the typical absence of explicitly stated (and complete set of) underlying assumptions made. While methods based on these theories can produce adequate results, the methods generally only work in a limited number of cases. This stems from the lack of a complete theory that guides the application or warns of inappropriate situations. Therefore, it is concluded that an approach to inexact probabilistic reasoning should be based on a mathematically sound theory. Such is Bayesian probability theory.

6.2 Proposed Methodology

In the following, a method for inexact probabilistic reasoning over graph-based models is introduced. This method is based on Bayesian probability theory and makes extensive use of Bayesian inference. To reason about graph-based models (which, by their definition from section 5.2, encode propositional statements), random variables with propositional target spaces are defined. Specifically, the elements in the target spaces are *graph patterns* (and, therefore, sets of *subgraphs*) that represent (conjuncted) *propositional statements* about (classes of) entities in the graph-based models. Inferring the probability of an event associated with a random variable therefore results in determining the probability of the truth of a corresponding proposition. This process is also known as *classification* in machine learning [149].

To set up a reasoning model, a sample space is defined and a set of random variables (and corresponding patterns) is elicited. Thereafter, a Bayesian network (see section 2.3.3) is constructed to make the (conditional and global) independence of variables explicit. Once the structure has been defined, beliefs on network parameters are elicited and represented by Dirichlet distributions (primarily for convenience (see section 2.3.5)). For each outcome to an experiment, *observations* about the outcome are made and *evidence* is collected in support or opposition of a *hypothesis*. This is done by polling the graph-based model by matching patterns associated with random variables, or by consulting external information sources. A hypothesis (or set of hypotheses) can either be determined by considering all unobserved random variables (i.e., those random variables for which none of the patterns associated with events lead to a match), or pre-defined (e.g., when the questions of interest can be defined a-priori, such as *"How probable is it that the outcome represents an inconsistent part of the graph-based model?"*). Once the hypothesis (or question of interest) has been defined, the Bayesian network can be utilized for inferring a probability of the hypothesis being true. This results in the (rational and sound) inference of a proposition (in the form of a subgraph) whose truth value is uncertain. This process is presented in more detail in the following.

6.2.1 Simplifying Assumptions

The proposed approach is an initial exploration into using a combination of Bayesian inference and pattern matching for abductive reasoning over graph-based models. Therefore, to demonstrate the value and (technical) viability of the proposed approach in a manageable form, its complexity is reduced through a number of assumptions.

Firstly, it is assumed that a mutually exclusive graph pattern can be identified for any of the events defined over the considered random variables. This subsumes that graph patterns can be identified for all defined events, and limits the associated propositions to statements describable by the graph-based model. The importance of the mutual exclusivity assumption is that the mapping of an outcome to the target space of each random variable must be unique. In other words, each random variable must be defined in such a way that every outcome of the experiment maps to exactly one value of the random variable (recall the definition of random variables as mappings from the sample space to respective target spaces).

Secondly, the assumption is made that only discrete random variables are considered when defining reasoning knowledge. The rationale for this is, primarily, a reduction in the expected computational complexity of probabilistic inference (recall from section 2.3.4 that inference in general Bayesian networks is NP-hard), so that the approach can be analyzed sufficiently within the scope of this dissertation.

Thirdly, the assumption is made that the random variables of interest for inference are known a-priori and are two-valued (i.e., binary) discrete random variables. This assumption is meaningful, since the inferences of interest for applications to inconsistency identification (*inconsistent* or *not inconsistent*, and *overlapping* or *not overlapping*) are all binary. Furthermore, most existing tools for evaluating such *classifiers* apply to binary classifiers only.

6.2.2 Illustrative Reasoning Scenario

To illustrate the general concepts underlying the proposed approach, an example often cited in the related literature on machine learning is adapted. This example stems from the field of medical research and is introduced in the following.

In medical research, Bayesian inference is often used as an abductive reasoning tool for performing *diagnostic* reasoning. A classical example from the literature is that of inferring the probability of a patient having lung cancer. Here, an adaptation of the example described in [152] is used. In the example, five (binary) discrete random variables are defined. The random variables with their respective (propositional)

Table 3: Example set of random variables with propositional target spaces used for diagnostic reasoning in medicine (adapted from [152]).

Variable	Propositional Target Space Values
H	*The patient has a history of smoking,* *The patient has no history of smoking*
B	*The patient has bronchitis,* *The patient does not have bronchitis*
L	*The patient has lung cancer,* *The patient does not have lung cancer*
F	*The patient experiences fatigue,* *The patient does not experience fatigue*
C	*The patient's Chest X-ray is positive,* *The patient's Chest X-ray is negative*

target spaces are depicted in table 3. Note that, as before, the convention is used that X represents the first event and $\neg X$ the second event.

By definition, the propositional values for the random variables are measurable (or testable) qualities or properties of a random outcome of an associated experiment. Here, the outcome of such an experiment is a (randomly selected) *patient* from a set of patients. A doctor has prior beliefs on all of the associated events: e.g., a doctor's belief on the event that a randomly selected patient has lung cancer may be expressed as $P(L) = 0.001$.

Independence assumptions among the random variables are made explicit using the Bayesian network depicted in figure 26. Note that the variables of interest for inference are depicted as vertices with colored background. It is assumed that all beliefs on network parameters have been determined in an elicitation process. For brevity, only the prior beliefs on L and B (i.e., the network parameters for L and B) are shown. This is justified by the fact that, for purposes of demonstrating the core concepts of the proposed method, it is sufficient to assume any values for the network parameters.

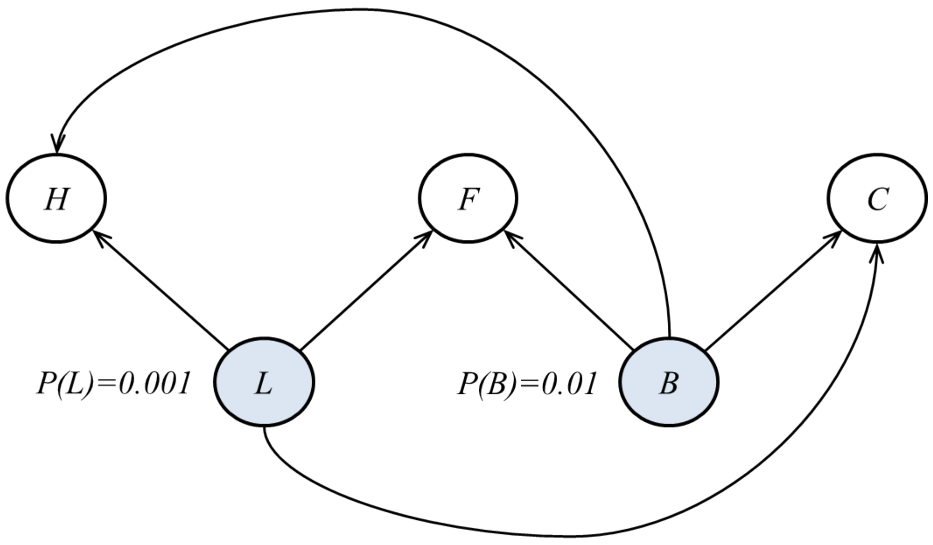


Figure 26: Bayesian network for the running example used in illustrating the proposed methodology. Vertices with colored backgrounds indicate variables of interest for inference. Note that only the beliefs on network parameters for L and B are given for brevity.

Given a randomly selected patient, a doctor gathers information *about* the patient by observing, testing or polling the patient. For instance, whether or not the particular patient has a history of smoking can be elicited from the patient directly. Since *any* patient either has, or does not have, a history of smoking (i.e., these are mutually exclusive events), the doctor is given information about the value of H for this particular patient. The value of F can be determined analogously. Additionally, whether or not the chest X-ray is positive can be determined by performing an associated test. It is assumed that a record of this patient data exists in the form of a graph-based model. An extract depicting a sample patient record as a graph-based model is illustrated in figure 27.

6.2.3 Using a Graph-Based Model as a Source for Information

Note that each of the propositional target space values in table 3 refer to a *patient*. Such a (randomly selected) patient was also described to be the outcome of the experiment. Epistemologically, the term *"the patient"* can be understood to represent

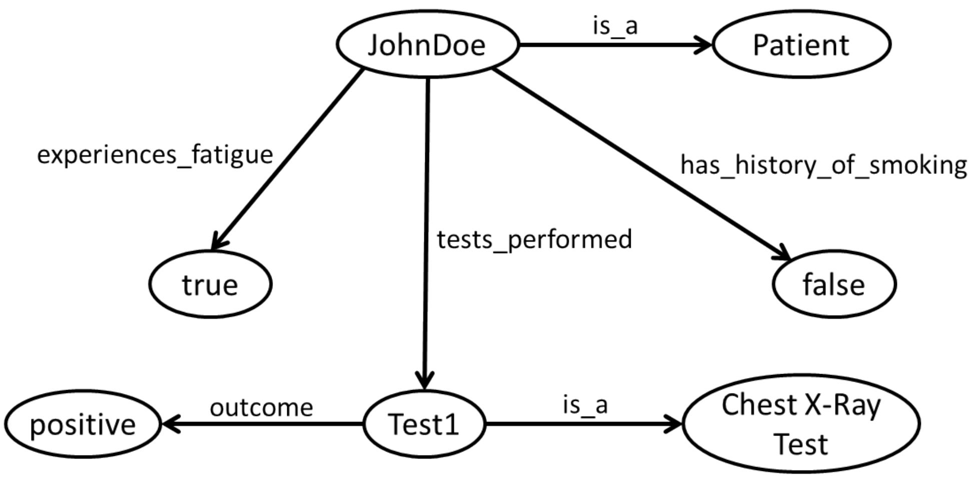


Figure 27: Sample patient record as a representative example of a graph-based model.

an individual from a class of *patients*. Therefore, the *sample space* for the conducted experiment consists of elements that represent patients.

Consider the graph-based model depicted in figure 27. Assuming an intuitive understanding of the relationships *is_a*, *outcome* and *tests_performed*, one can easily extract information about the individual *John Doe* who *is a patient*, and can be regarded as a single outcome of the probabilistic experiment. Now, one can use the Bayesian network from figure 26 to determine the probability of *John Doe* having lung cancer by updating the prior belief about *any* patient having lung cancer $P(L) = 0.001$ with the observations made about *John Doe*:

$$P(L \mid F, \neg H, C) = \frac{P(L)P(F, \neg H, C \mid L)}{P(F, \neg H, C)} = \frac{P(L, F, \neg H, C)}{P(F, \neg H, C)}$$

Using the methods introduced in section 2.3.4, this probability can be inferred (computationally) from the provided Bayesian network. The interesting question is now: how would a computer perform these *observations* – i.e., how can the evidence be collected computationally – to determine *John Doe*'s probability of lung cancer? Recalling the methods introduced in chapter 5, such information can be extracted from a given graph-based representation of a model by means of *graph pattern matching* and *querying* (see section 5.3).

Formally, the validity of this can be argued using the definition of a random variable through its pre-image. Recall from section 2.3.1.3 that random variables are mappings from the set of outcomes to a measurable target space – e.g., $X : \Omega \to E$, where X is a random variable, Ω is the sample space and E is a target space. The preimage of a random variable X is then defined as:

$$X^{-1}(x) = \{\omega_i \in \Omega \mid X(\omega_i) = x\}$$

Here, $X^{-1}(x)$ is defined by the *set of outcomes* (that is contained in the σ-algebra $\mathcal{F}$) – i.e., an *event* – for which X maps to x. ω_i is an outcome and, therefore, $\omega_i \in \Omega$. As a concrete example, consider the random variable C, which takes on one of two values for a random outcome: *"The patient's Chest X-ray is positive"* or *"The patient's Chest X-ray is negative"*. The events are *mutually exclusive* and, by definition of a random variable, *any* outcome of the experiment must map to either value. Let $c_0 =$ *The patient's Chest X-ray is positive* and $c_1 =$ *The patient's Chest X-ray is negative*. Then:

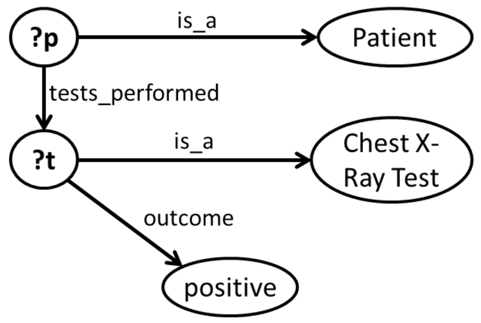
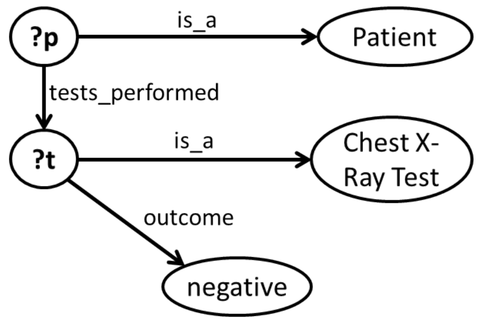
$$C^{-1}(c_0) = \{\omega_i \in \Omega \mid C(\omega_i) = c_0\} = \Omega \setminus C^{-1}(c_1)$$

John Doe is defined as one possible patient selected at random (i.e., one possible outcome). Let the outcome denoting *John Doe* be defined as ω_{JD}. One possible source of information about *John Doe* is the graph-based model depicted in figure 27. Using this graph-based model, one can determine that (among other information):

$$C^{-1}(c_0) = \{..., \omega_{JD}, ...\}$$

In other words, it is known from the information available in the graph-based model that $C(\omega_{JD}) = c_0$. Now consider the graph patterns depicted in table 4. Let the graph pattern associated with the event *The patient's Chest X-ray is positive* be depicted by $\mathbf{P}_{C=c_0}$, and the sample graph-based model be depicted by $\mathbf{G}$. Then, a set of matches $M_{\mathbf{P}_{C=c_0},\mathbf{G}}$ to the graph pattern $\mathbf{P}_{C=c_0}$ is the set of sets of triples depicting subgraphs

Table 4: Target space values (possible events) and associated graph patterns for the random variable C.

Event	**Associated Graph Pattern**
The patient's Chest X-ray is positive	 ?p is_a Patient tests_performed ?t is_a Chest X-Ray Test outcome positive
The patient's Chest X-ray is negative	 ?p is_a Patient tests_performed ?t is_a Chest X-Ray Test outcome negative

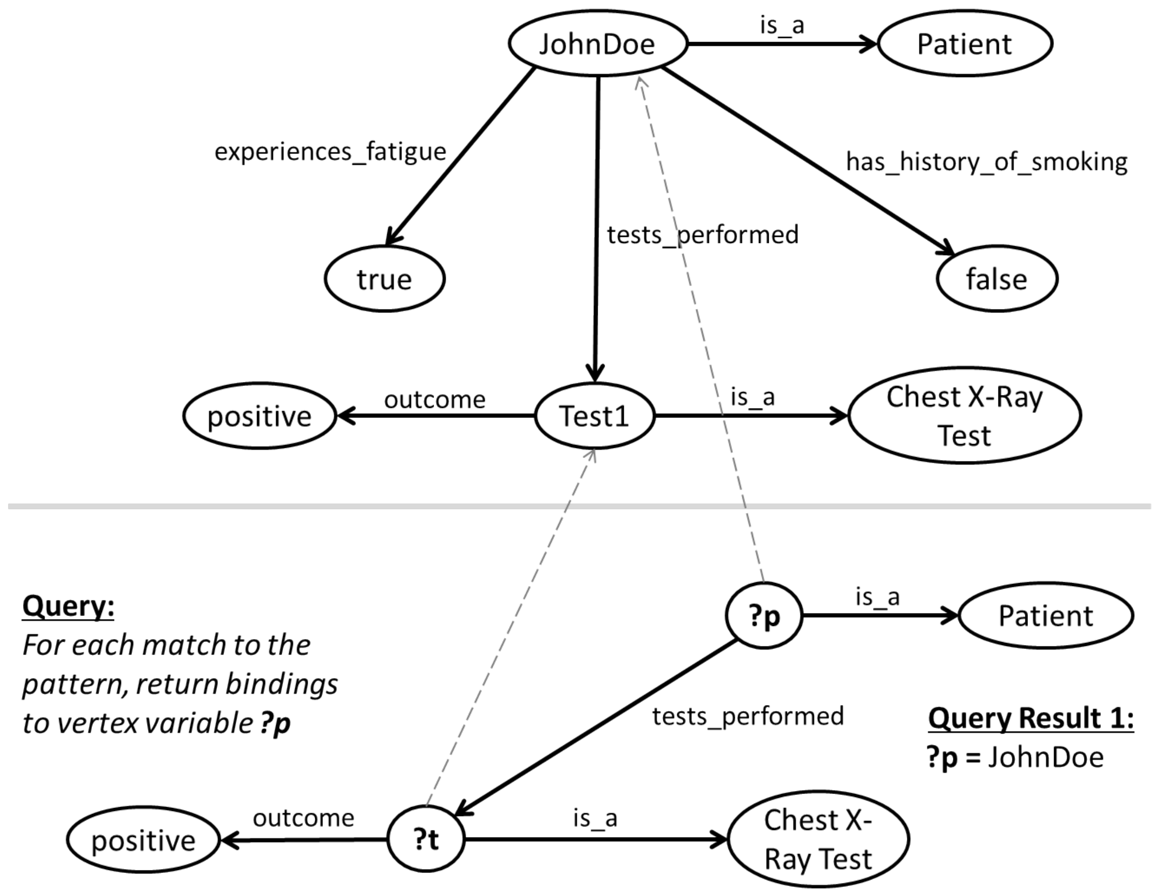


Figure 28: Query for retrieving information about *John Doe*.

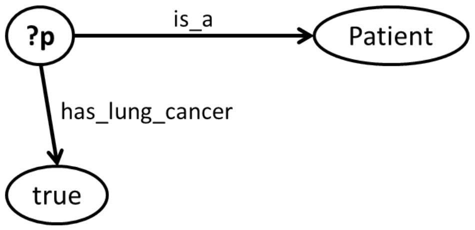
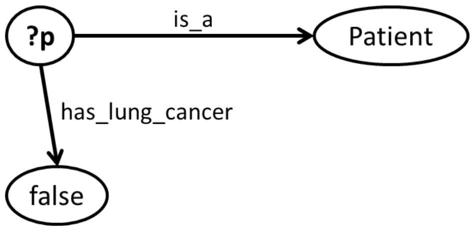
of the graph-based model which describes patients and their relation to a positive outcome on a chest X-ray test. Formulating this as a query (see section 5.3), one can retrieve the bindings to the variable vertex depicted by **?p** for each match, each of which denotes a patient, and, therefore, an outcome. The set of all bindings to **?p** in the graph pattern is then the set of outcomes which are known to be elements of the pre-image of $C = c_0$. Repeating this process for other random variables leads to sets of observations about patients. Note that the variable bindings in a set of observations about each patient must be consistent (i.e., the same). Also note that, in order to refer to a patient (and, the same patient in each case), a common *base pattern* is required that is shared by all patterns. Here, this base pattern is (**?p**, *is_a*, *Patient*).

Note that, depending on the information available in the graph-based model, and depending on the definition of the sample space, the result of such a query will only return a subset of the respective pre-image. It is tempting to define one pattern for event c_0, and to assume that all patients that are not returned as a result of matching the pattern in the graph must therefore be elements of $C = c_1$. However, this assumes knowledge about the patient that is neither justified nor supported by the information source. Instead, one should treat a case where a patient is in neither set as a case where *it is unknown whether the patient is a part of either pre-image*. Formally, this is equivalent to saying that *using the given means, and using the graph-based model as the only source of information, it is impossible to determine with certainty whether the patient is a part of the event $C = c_0$ or $C = c_1$*, and such signifies a state of incomplete information. This does not mean that it cannot be determined: for instance, a doctor may have not performed a chest X-ray test on a particular patient because information about the outcome of the test may not have *significantly influenced* the probability of lung cancer (or bronchitis), given the other observations. When computing a posterior probability in such cases, the variable is simply marginalized. Note that this was already done for the variable B (which denotes whether a patient has bronchitis or not) when $P(L \mid F, \neg H, C)$ was inferred from the Bayesian network.

6.2.4 Storing Results of Abductive Inference

Once information about an outcome has been received, prior beliefs can be updated. To illustrate this, suppose that it was determined (using appropriate patterns) that *John Doe* (1) has no history of smoking, (2) experiences fatigue, and that (3) the result of a chest X-ray test is positive. This information can now be used to update the prior belief about *John Doe* having lung cancer. That is, one can now determine the posterior belief $p(L \mid \neg H, F, C)$. This posterior distribution can be inferred using the

Table 5: Target space values (possible events) and associated graph patterns for the random variable L.

Event	**Associated Graph Pattern**
The patient has lung cancer	
The patient does not have lung cancer	

Bayesian network and computed using the inference methods for Bayesian networks introduced in section 2.3.4.

The result of an abductive reasoning process over the graph-based model is a *propositional statement* (e.g., *John Doe has lung cancer*) whose truth value is uncertain. This inferred proposition is produced by *instantiating* the associated pattern. This is similar to how production (see section 2.2.3) and graph inference rules (see section 5.4) produce concrete inferred propositions (by instantiating a pattern that represents the consequent of such a rule). Table 5 contains patterns for the events *The patient has lung cancer* and *The patient does not have lung cancer*. By creating a set of triples in which the variable vertices are replaced with the mappings common to these variables in all patterns, a new instance is created. For example, given a match to the pattern associated with the event *The patient's Chest X-ray is positive*, the variable binding of **?p** and **?t** can be determined. To instantiate either of the patterns from table 5, this same mapping to the data graph for the common variable

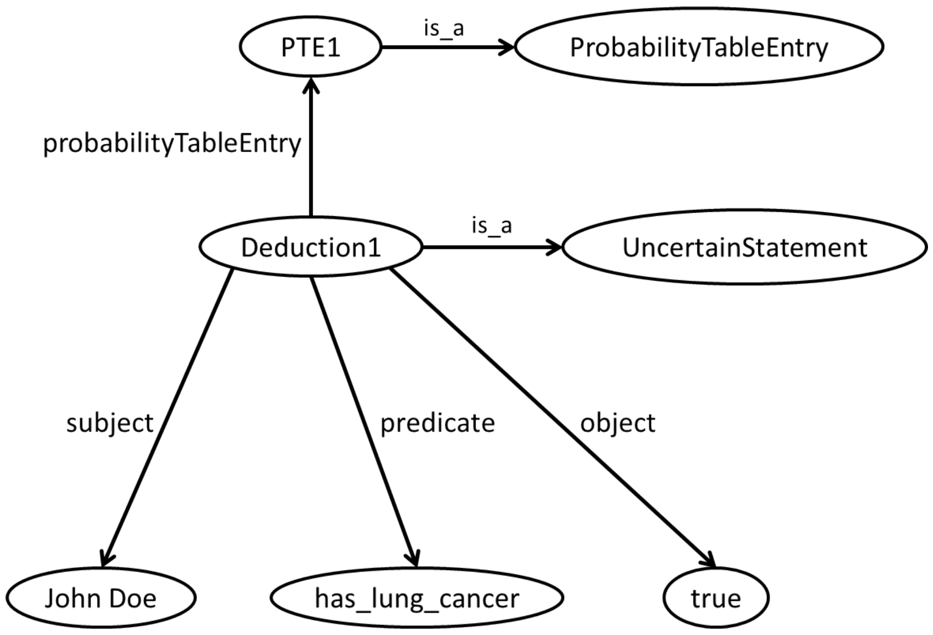


Figure 29: Illustration of how the inferred uncertain proposition *John Doe has lung cancer* is stored as a reified statement, and as part of the graph-based model.

?p is utilized.

One difference to inference rules where the consequent is accepted to be either *true* or *false* is that, here, a probability is associated with the truth value of the proposition. Therefore, one no longer has just a *(subject, predicate, object)* triple, but a quadruple *(subject, predicate, object, probability)*. To represent this in the graph-based formalism introduced in chapter 5, the concept of *reification* is used. Reification entails creating a vertex in the graph that denotes an *Uncertain Statement*, which points to a subject, predicate and object in the graph, and, in addition, points to a corresponding probability. This is illustrated in figure 29 for the proposition *John Doe has lung cancer*

Note the use of the predicate *probabilityTableEntry* in figure 29. Naïvely, one could have simply pointed to a numeric value representing the probability of the proposition being true. However, doing so results in a loss of information about *what the probability value represents* and on *what evidence it was based*. That is, the *argumentation chain* is missing. Therefore, the entry in a probability distribution (i.e., one particular mapping of an associated *probability mass function*) is pointed to instead. Such entries have a reference to the *value* of the random variable and a probability. Collectively, these entries define a *Distribution*, which is associated with an *(Inferred) Belief*. Such beliefs are *on* one or more *Random Variables*. Information used in updating a prior belief on the respective random variable is made explicit by pointing to *Events* using the predicate *given*. These events are associated with random variables and, collectively, define all events over random variables.

The devised method for representing the full deduction of an uncertain statement and the associated probability is exemplified for the simpler case of $P(L \mid \neg H)$ in figure 30. Note the inclusion of an *influences* statement between L and H. By including such relations among random variables, the Bayesian network is fully represented by the graph-based model as well. Note that, for each uncertain statement, only the *most informed state* should be stored. In other words, for each outcome, all of the observations available about the outcome should be incorporated. It would be irrational to do otherwise [18].

Note that the number of possible distributions stored as part of the graph-based model is finite, and is defined by the Bayesian network. By storing not only a proposition with an uncertain truth value and the corresponding probability, but also the evidence that was used in computing a posterior belief from a prior belief, the full rationale is captured also.

In table 6, the vocabulary used for constructing uncertain (reified) statements and storing associated probabilities is summarized. Note that the vocabulary is generally applicable for discrete random variables. As mentioned in section 6.2.1, the use of discrete random variables is a simplifying assumption. The vocabulary must be extended if continuous variables are to be included.

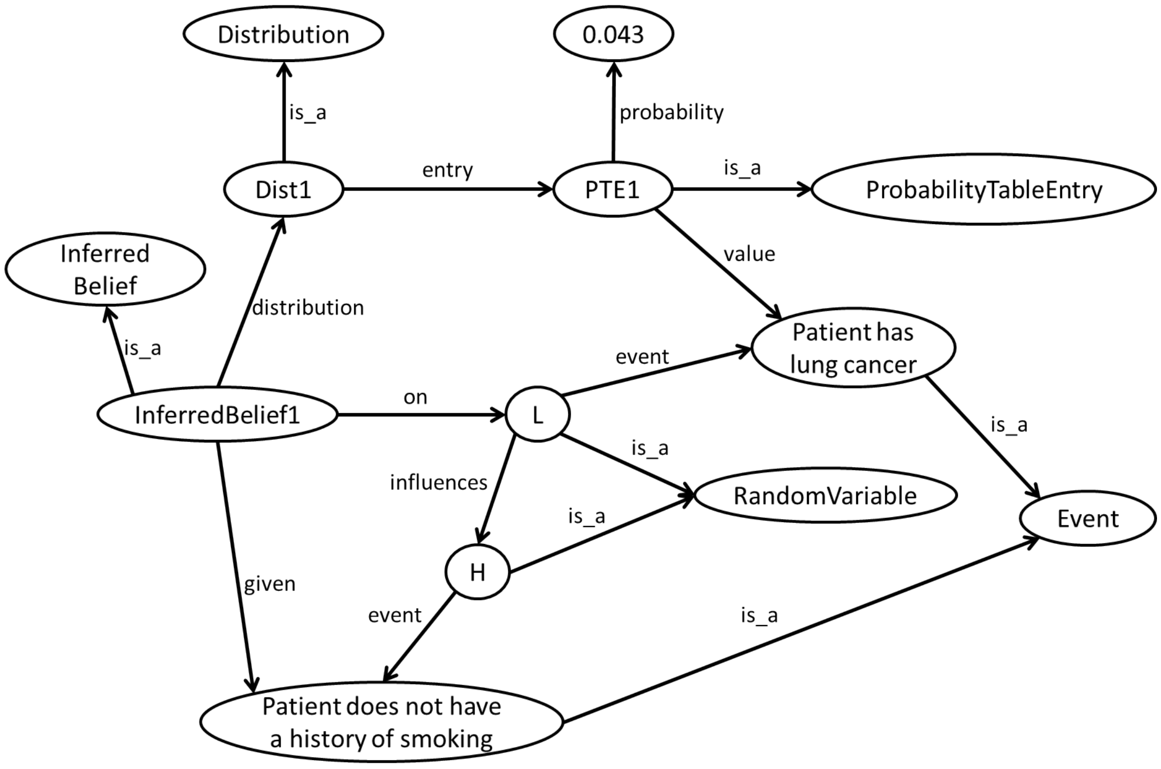


Figure 30: Depiction of how the inferred probability distribution for $P(L \mid \neg H)$ is represented by a graph-based model. Note the influence relationship between random variables L and H: together, all random variables and influence relationships between random variables implicitly represent the Bayesian network.

Table 6: Vocabulary used for storing propositions with uncertain truth values and the corresponding argument chain.

Vocabulary Term	**Description**
UncertainStatement	An uncertain statement.
subject	Pointer to the subject of an uncertain statement.
predicate	Pointer to the predicate of an uncertain statement.
object	Pointer to the object of an uncertain statement.
probabilityTableEntry	Pointer to a *ProbabilityTableEntry* instance.
RandomVariable	A random variable.
Event	A value of a random variable, leading to the formation of an event.
(Inferred)Belief	An (inferred) belief.
Distribution	A probability distribution, defined by one or more *ProbabilityTableEntry*s.
ProbabilityTableEntry	An entry in a probability distribution, defined by the *value* of an associated random variable and a *probability*.
probability	Predicate used for associating a *ProbabilityTableEntry* with a probability.
value	Predicate used for associating a *ProbabilityTableEntry* with a value of a random variable.
entry	Assigns a *ProbabilityTableEntry* to a distribution (more than one per *Distribution* possible).
event	Assigns an object of type *Event* to a *RandomVariable*.
on	Defines the random variable that the belief is on.
given	Pointer used in denoting the evidence used in computing the distribution associated with an *(Inferred)Belief*.
distribution	Predicate used in assigning a distribution to an *(Inferred)Belief*.
influences	Influence relationship between *RandomVariable*s.

6.2.5 Gathering Additional Evidence from External Sources

In the previous sections, an abductive reasoning method based on Bayesian inference and pattern matching is introduced for the retrieval of supporting (and opposing) evidence for a given hypothesis from a graph-based model. It is concluded that pattern queries can be used for retrieving information about individual outcomes of an associated experiment, but noted that one cannot always determine *all* possible information. That is, if information about a particular outcome (in the given example, a *patient*) allowing one to determine the value of a particular random variable is simply missing from the graph-based model, it cannot be determined using the introduced pattern matching based method alone. Indeed, it may not even be valuable to *query* a particular pattern in the first place if the cost associated with executing the pattern query exceeds the benefit gained from receiving additional information. This argument will be built upon in the following by considering additional information sources – that is, means of retrieving information outside the realm of the single graph-based model considered so far.

One way to quantify these thoughts is presented in the following and is based on *Value of Information* (VoI) theory. In general, it is rational to always use all of the available information. However, acquiring information in addition to what is readily available (i.e., known) invokes a cost. For the case of querying graph-based models, this cost is the cost of executing a query / matching a pattern. In the following, assume that a set of information sources is available, as well as methods for retrieving information from these. Say a decision D is to be made. For illustrative purposes, let D represent the decision by the doctor to inform the patient that he or she has been diagnosed with lung cancer. Say that the decision is made based on information k (which may be any combination of evidence considered in the previous sections). For illustrative purposes, let this information k represent the fact that the patient is known to have a history of smoking and is experiencing fatigue. Let j depict a quantity that provides additional *perfect* (that is, certain) information which may influence (that is, potentially change) D, and let i be a source for this information. Let j represent the outcome of the chest X-ray test, and i the X-ray procedure itself. The expected value of information EVI for retrieving information about j is then:

$$EVI = \mathbb{E}\left[V(D_{post-j})\right] - \mathbb{E}\left[V(D_{pre-j})\right]$$

Here, $V(D_{post-j})$ and $V(D_{pre-j})$ denote the value (or payoff) of the decision made after and before acquiring information about j, respectively. Therefore, the expected value of information is the difference in expected values of a decision made in either state of information. Acquiring information about j – that is, performing a test required to retrieve information about j – invokes a cost $C_{r_{ji}}$ of retrieving information about j from information source i. Within the context of the illustrative example, $C_{r_{ji}}$ is the cost associated with performing the X-ray procedure. Therefore, one should only invest this cost $C_{r_{ji}}$ if j influences the decision to such an extent that the decision changes, and the benefit (value) *gained* from this exceeds the cost. If j does *not* change the decision, or the cost exceeds the gain in value, retrieving information about j adds no value and i should not be polled for j.

Note that, within the context of the method proposed in this chapter, decisions D may be considered *classification* decisions. As will be demonstrated in more detail in section 7.3.1, such classifications include deciding whether or not an inconsistency is to be reported to a user. There, decisions, and the corresponding expected value take into account the (true) probability of *making a wrong decision* and the associated consequences (invoked costs).

For practical purposes, it can be assumed that the cost of retrieving information from the graph-based model by means of pattern matching is very low compared to its benefit. Note that this does not necessarily represent an accurate assumption for the general case (e.g., for very large graphs, and a given pattern matching algorithm, retrieving information may be very expensive (computationally)). Also, in some cases, where classification decisions are made that have a high impact (such as determining whether an inconsistency is present), it may be valuable to poll external information sources such as a human. Since these are considerations that are only meaningful to explore in specific contexts, this will be discussed further and more concretely in chapter 7, where the approach presented in this chapter is applied to identifying inconsistencies.

6.3 Algorithmic Implementation

In the following, an algorithmic implementation of the abductive reasoning procedure described in the previous section is presented. The algorithm is intended for an *exhaustive* application (rather than simply querying for observations about a single outcome of a single run of an experiment) and is robust to the extent that outcomes do not have to be a single entities (such as a *patient*) but can be more complex constructs, such as pairs of entities (e.g., pairs of patients). A number of simplifying assumptions are made:

- The expected cost of matching any pattern associated with a given Bayesian network is negligible compared to the expected benefit gained.
- All patterns associated with the Bayesian network contain a common sub-pattern that identifies the outcome of an associated experiment.
- The inference is applied *exhaustively* to all entities from a specified (sub-)graph that can be identified as possible outcomes of an associated experiment.

The first assumption can be argued to be valid *on average*. The assumption simplifies the reasoning process considerably by not requiring the decision process (i.e., the classification process) to be incorporated into the algorithm itself. This classification decision may be based on a heuristic, or require input from a human, and is likely to vary between application scenarios. The second assumption is formally correct if it is assumed that the algorithm simply performs a sufficient number of trials to select each identifiable outcome in the provided (sub-)graph at least once.

6.3.1 Algorithm Overview

Two primary operations need to be carried out by an algorithmic implementation of the abductive reasoning method introduced in section 6.2: pattern matching in directed multi-graphs, and inference in a Bayesian network. As mentioned in section 5.3.2, pattern matching is NP-complete, and, as mentioned in section 2.3.4, inference in Bayesian networks is NP-hard.

Given the complexity of these operations, an *incremental algorithm* (see algorithm 1) is proposed. Specifically, this means that only the *changes* made to an input graph are considered. For simplicity, and because this is a first exploration of such an algorithm, the incremental behavior of algorithm 1 is only valid for *additive* changes to the graph. A *deletion* or *modification* of an existing statement requires a complete re-evaluation of the algorithm over all relevant triples. Four inputs are provided to the algorithm: a data graph $\mathbf{G}$ (i.e., the graph-based model acting as a source of information), a set of triples T added to the graph, a Bayesian network $B = (\mathbb{G}, P)$, and a deductions graph $\mathbf{G}_D$ where all deductions are stored. Note that T is the set of all triples in $\mathbf{G}$ if all deductions about all identifiable outcomes in the graph-based model are to be re-evaluated. $T \subset \mathbf{G}$ if the incremental behavior of the algorithm is made use of, or if the algorithm is to be evaluated over only a subset of the data graph. Also note that all deductions are removed – i.e., $\mathbf{G}_D$ is cleared – if existing triples in $\mathbf{G}$ are modified or deleted. A preliminary version of this algorithm has been published by the author of this dissertation in [107].

Verbally, algorithm 1 (in conjunction with algorithm 2) performs the following actions: for each triple t from T, observations *local* to t are stored in a map. Local, in this context, means that, for each of the patterns associated with the target space

Algorithm 1: Infer propositions with uncertain truth values given a data graph, a set of changes to the graph, a Bayesian network, and a deductions graph.

```
Algorithm doInference(Graph $\boldsymbol{G}$, Triples $T$, BayesNetwork $B$, Graph $\boldsymbol{G}_D$)
    for $t \in T$ do
        for $rv \in B.RandomVariables$ do
            // Retrieve information about rv from the graph in the context of $t$
            Obs[outcome] $\longleftarrow$ Obs[outcome] $\cup$ observe($t$, rv, $\mathbf{G}$) ;
        end
    end
    Outcomes $\longleftarrow$ Observations.Keys ;
    // Iterate through the observations about each identified possible outcome
    for $outcome \in Outcomes$ do
        for $observation \in Obs[outcome]$ do
            // Compute a list of random variables about which an observation
               was made
            ObservedRVs $\longleftarrow$ ObservedRVs $\cup$ observation.RandomVariable ;
            // Store this observation as a tuple consisting of a random
               variable and the observed value
            m $\longleftarrow$ (observation.RandomVariable, observation.Value) ;
            Measurements $\longleftarrow$ Measurements $\cup$ m ;
            // Compute the union of all (variable) bindings
            AllBindings $\longleftarrow$ AllBindings $\cup$ observation.match.Bindings ;
        end
        // The unobserved random variables are those of interest for inferring
           propositions with uncertain truth values
        UnobservedRVs $\longleftarrow$ B.RandomVariables $\setminus$ ObservedRVs ;
        for $rv \in UnobservedRandomVariables$ do
            // Retrieve any related prior deduction (note that in the event of
               deleting, or modifying existing statements in the graph, the
               deductions graph is cleared)
            prior $\longleftarrow$ retrieveBeliefAbout(rv, AllBindings, $\mathbf{G}_D$) ;
            // Additionally consider newly found evidence
            inf $\longleftarrow$ B.updateBelief(rv, AllBindings, Measurements, prior) ;
            // Create graph-based representation of deduction
            $\mathbf{G}_{SD}$ $\longleftarrow$ updateDeduction(rv, AllBindings, inf) ;
            // Add to deductions graph
            $\mathbf{G}_D \longleftarrow \mathbf{G}_D \cup \mathbf{G}_{SD}$ ;
        end
    end
    return $\mathbf{G}_D$
```

Algorithm 2: Using a triple t as a hook, retrieve a set of matches to each of the patterns associated with the specified random variable. Return a map with a reference to the outcome as the key, and a set of tuples as the entry, each of which consists of a subgraph (representing a single match to the pattern), bindings (including variable bindings), and random variable value.

```
1  Algorithm observe(Triple t, RandomVariable rv, Graph G)
       // Iterate through each target space value
2      for value ∈ rv.Values do
3          pattern ⟵ value.AssociatedPattern ;
           // Attempt to match t against the current pattern
4          TBindings ⟵ match(t, pattern) ;
5          if TBindings ≠ ∅ then
               // If successful, use t as a hook by binding a part of the pattern
                  to t, and find all possible matches to the pattern in the
                  context of t
6              Matches ⟵ findMatches(pattern, TBindings, G) ;
               // Iterate through the set of matches
7              for match ∈ Matches do
                   // Extract the outcome from the match by extracting the
                      bindings to the common sub-pattern
8                  outcome ⟵ extractOutcome(match) ;
9                  observation ⟵ (match, rv, value) ;
                   // Store the specific observation made about rv by associating
                      it with the outcome
10                 Obs[outcome] ⟵ Obs[outcome] ∪ observation ;
11             end
12         end
13     end
14     return Obs
```

values of the random variables, an attempt is made to match t against the pattern. If this succeeds, one part of the respective pattern is bound to t. Recall that t is a triple in the data graph $\mathbf{G}$. If the pattern consists of only one triple, then all possible matches to the pattern in the data graph $\mathbf{G}$ are comprised of the single match to t. However, if the pattern is more complex, there may be more than one match to the pattern in the context of t. For example, consider the situation depicted in figure 31. There, the triple t matches the provided pattern *partially* – that is, **?p** binds to *JohnDoe*, and the constant vertices *is_a* and *Patient* map to the respective label-equivalent vertices. However, a full match to the pattern requires additional bindings. These are identified in the data graph (note that the triple t is intended to *reference* a part of the data graph). In the situation depicted, the variable **?t** would, for instance, bind to *Test1* for one particular match. If the data graph contains *multiple* chest X-rays for *John Doe*, then multiple matches become part of the set computed in line 6 of algorithm 2 (which, here, by definition of the associated random variable, would be illegal unless both matches represent semantically identical concepts[3]).

For each of the matches, the associated outcome is extracted, and by the assumptions made, all patterns associated with the Bayesian network share a common sub-pattern identifying a possible outcome of the associated experiment. In figure 31, this common sub-pattern is the triple (**?p**, *is_a*, *Patient*). Recall from the examples given in section 6.2, and specifically from table 5, that the patterns associated with the different target space values all shared this triple. Given information about the associated outcome, it can now be claimed that one particular observation has been made about the particular outcome. Before this observation is used, it is stored in a map, which stores a set of observations and associates this set with a particular outcome.

Once all observations have been computed, one can easily determine for which

[3]This, and other related issues are investigated in more detail in chapter 7.

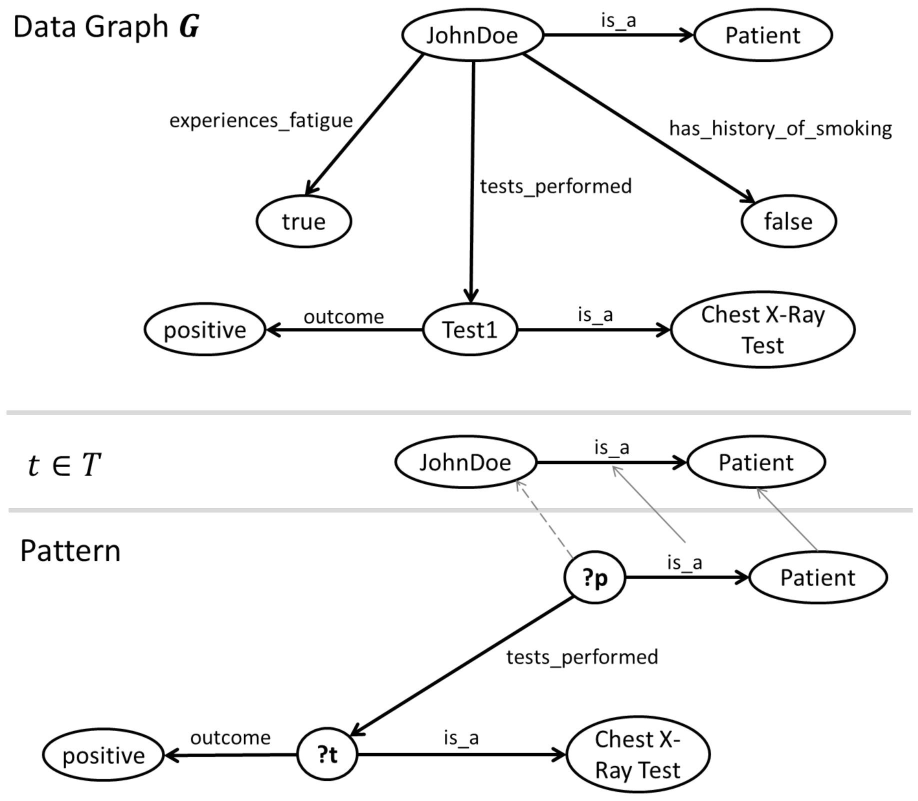


Figure 31: Illustrative example of a partial pattern match: matching a single triple t in a graph pattern.

random variables *no* information was available in the graph-based model. To illustrate this using the running example, this means that for the outcome *John Doe* (i.e., ω_{JD}), all observations available in the data graph (e.g., that the chest X-ray test outcome was positive) are stored in the set $\text{Obs}[\omega_{JD}]$. However, whether *John Doe* has lung cancer or not, or bronchitis or not, is not stored in the data graph. Such unobserved random variables are computed through intersection of the set of all random variables in the Bayesian network and the set of observed random variables (see line 15 of algorithm 1). Each of the patterns associated with the target space values of the unobserved random variables can now be instantiated using the bindings from the observed patterns (given that this set of bindings is sufficient for instantiating the pattern), and a posterior probability can be computed from a prior belief on the random variable and the observed evidence. Once again, this probability can be inferred from the Bayesian network. Finally, a graph-based representation of the deduction is constructed according to the rules described in section 6.2.4.

6.3.2 Handling the Arrival of New Information

As outlined in the previous section, the proposed algorithm is of an *incremental* nature. This means that, once triples are added to the data graph, any new observations are added to previously made observations, and the probability is re-computed. However, as mentioned previously, once existing statements are removed from the data graph (or modified), a full re-evaluation is required. This is illustrated in algorithm 1 on line 17, where evidence collected previously is retrieved. This previously collected information is taken into account when calculating the posterior on line 18. Note that if the new information leads to a situation in which there are two different observations about a random variable, it is indicative of either an inconsistency in the model, or an incompleteness of the pattern. These issues are discussed in more detail in the next chapter.

6.4 Summary

In this chapter, a generic method for inexact (abductive) probabilistic reasoning over graph-based models is presented. The method is based on concepts from Bayesian probability theory (such as Bayesian inference) and pattern matching in graphs. Three main aspects are covered in this chapter: the first part reviews various inexact reasoning methods from the literature and provides rationale for using a Bayesian approach. The second part introduces the proposed method in detail. Finally, the third part presents algorithmic procedures for implementing the approach in practice.

As an argument for using Bayesian probability theory, two well-researched approaches to inexact (probabilistic) reasoning are reviewed and compared to Bayesian probability theory: certainty factor theory (CF theory) and Dempster-Shafer theory (DS theory). While both methods have been applied with success in practice, a concern related to their formality and soundness is raised. It is argued that the restricted applicability of the methods is a reason for choosing Bayesian probability instead. However, while acceptedly the most formal approach, potential issues with regards to the elicitation of the required reasoning knowledge are identified and acknowledged.

The second part of the chapter introduces the proposed approach to inexact (abductive) probabilistic reasoning over graph-based models. The basis for representing, retrieving and manipulating the underlying graph are the concepts from chapter 5. In the approach, graph-based models are treated as a source of information. Observations about outcomes (defined by a *base pattern*) are made by evaluating matches to graph patterns. These graph patterns are associated with the target space values of random variables. Graph patterns associated with the same random variable must necessarily (by definition of random variables) lead to mutually exclusive outcomes. Bayesian networks are proposed as compact representations of the joint probability distribution over the random variables, and for the purpose of efficiently inferring probabilities. Using the Bayesian network, a posterior probability can efficiently be calculated from a prior and a set of evidence (i.e., observations). Based on the observations made, updated beliefs about unobserved events are calculated, and the associated patterns instantiated in the graph using the variable bindings from the base pattern and observations. These inferred propositions are then stored as reified graph triples with a reference to the probability of the truth value of the represented statement.

The third part of this chapter introduces algorithmic procedures for a practical implementation of the approach. The introduced algorithms are *incremental* in nature due to the expected complexity of the approach (based on the theoretical complexity of the underlying methods used). A re-evaluation of all deductions is only necessary in certain circumstances.

CHAPTER VII

BAYESIAN INCONSISTENCY IDENTIFICATION

In previous chapters, a foundation is laid for reasoning over heterogeneous models (chapter 5), and a method for inexact probabilistic reasoning is introduced (chapter 6. This is done in an effort to support the concretization and evaluation of the framework to inconsistency identification proposed in chapter 4. The aim of this chapter is to discuss the application of the developed concepts and methods for identifying (probable) inconsistencies in heterogeneous models.

The chapter is outlined as follows: first, the characteristics of an inexact probabilistic (abductive) approach to reasoning about inconsistencies are introduced, and the characteristics of the associated reasoning knowledge are detailed. The secondary, but nonetheless important, aim of this first section is to discuss the impact of such an approach on the life-cycle of a system. Thereafter, methods are suggested for acquiring and eliciting the required knowledge for reasoning about inconsistencies. Important special facets of eliciting inconsistency identification knowledge, such as eliciting beliefs about highly unlikely events, are detailed. Part of the discussion includes how making certain assumptions can, in certain cases, lead to a significant reduction in complexity. Aspects of reusability of the inconsistency identification knowledge are also briefly outlined. The last part of this chapter is concerned with the interpretation and presentation of the inference results.

7.1 *Inexact Inference of Inconsistencies*

From a deductive reasoning perspective, an inconsistency is (provably) present if it can be shown that some statement A and its negation are both true. In chapter 4, this view on inconsistencies was generalized to define an inconsistency as a state of conflict where *sufficient evidence* exists to conclude that an inconsistency is present. This view is supported by the fact that, for the case of reasoning across a set of disparate, heterogeneous models, an underlying formal system capable of describing the result of the composition of the various models is non-trivial (if not impossible) to identify (in practice).

In the following, the characteristics of inconsistencies and reasoning knowledge used in identifying inconsistencies within the proposed abductive inference framework are detailed. Additionally, the process of inconsistency identification, and acquisition and refinement of the required reasoning knowledge is presented to illustrate the role and impact of inconsistency identification on the life-cycle of a system. Finally, assumptions about the supporting infrastructure are made explicit.

7.1.1 Inconsistency Identification Knowledge

The section starts with a brief definition of *inconsistency identification knowledge*. Inconsistency identification knowledge is considered the agglomeration of all knowledge utilized for the process of identifying inconsistencies. Specifically, this knowledge includes:

- A description of the class of the entity (or entities) associated with an instance of a particular type of inconsistency (i.e., the definition of the *outcome* of an experiment)
- The definition of the structure and parameters of a Bayesian network
- Patterns associated with the target space values of the random variables in the Bayesian network

While arguably only a prerequisite, the knowledge required for translating formal models to the proposed common representational formalism (section 5.2), mediation (section 5.5), and inference rules enabling the (partial) interpretation of the graph-based models (e.g., to calculate the transitive hull for some predicates) should also be considered a part of inconsistency identification knowledge. This is due to the inherent dependence of the inexact reasoning method on the underlying representational formalism. Note that the degree to which these inference rules need to be defined *externally* (i.e., external to the patterns associated with a Bayesian network) depends on the expressiveness of the concrete pattern language used. Some pattern formalisms allow for basic inferences to be incorporated as a part of the pattern (e.g., by allowing for edge production through the use of regular expressions (see, e.g., the concept of *property paths* defined as part of the pattern formalism for the query language of SPARQL 1.1 [226])). Therefore, the following is explicitly considered a part of inconsistency identification knowledge also:

- Transformation definitions for translating formal models to produce representations in a concrete implementation of a graph-based representational formalism
- Mediation rules allowing for the semantic abstraction of the produced graph-based models
- Graph inference rules for interpreting the semantics of models at query time

An accompanying infrastructure allowing for the interpretation of the inconsistency identification knowledge is subsumed, but considered outside the definition of inconsistency identification knowledge.

7.1.2 Characteristics of Abductively Inferred Inconsistencies & Related Reasoning Knowledge

As introduced in section 2.2.3.2, abduction is an *explanatory* view on inference, the result of which is the *best explanation* for a set of observations made. Abductive inferences are not necessarily logically correct. That is, they cannot be *proven* to be logically correct. In the proposed framework, this is considered an advantage rather than a disadvantage, since it allows reasoning with incomplete and abstract information and knowledge. However, the reasoning knowledge used for applying the method introduced in chapter 6 must be carefully crafted to avoid erroneous, or even inconsistent states of the reasoning knowledge itself.

The proposed Bayesian framework considers discrete random variables with different propositional target spaces. By definition of a discrete random variable, the values for a random variable must be *mutually exclusive* and *exhaustive*. That is, for every possible outcome to an experiment, the random variable must take on exactly one value given a mechanism for measuring the random variable, and sufficient information. Since, in the proposed framework, values of random variables are associated with patterns, which in turn are used as one possible mechanism for measuring the random variable, the interesting question to consider is: what happens if, for a given outcome, more than one pattern associated with the same random variable results in a match? This would be in contradiction with the definition of a random variable.

There are two possible sources for such erroneous states: the graph-based model being reasoned over, and the reasoning knowledge (here: inconsistency identification knowledge) itself. If the source is the reasoning knowledge, it is a likely indication that the reasoning knowledge is either inconsistent or incomplete. Proving the inconsistency of reasoning knowledge is difficult at best: an intuitive definition of inconsistent reasoning knowledge is that, given an inconsistency free graph-based model, two mutually exclusive events can be identified. However, whether this truly marks a state of inconsistency, requires a proof that the patterns used in identifying the mutually exclusive states are necessary and sufficient conditions for these states. If this is not the case, the reasoning knowledge is likely *incomplete* or *underspecified*, which requires *refinement of the knowledge*. Indeed, this could also be an indicator that the values are not exhaustive. If the reasoning knowledge itself is (provably) *not* inconsistent, it is likely that the graph-based model itself is inconsistent, since it describes two mutually exclusive states.

These observations about the reasoning knowledge utilized in the proposed approach are important, since they highlight the necessity of making the assumption that the reasoning knowledge is sufficiently free of inconsistencies and complete, unless it can be demonstrated otherwise – that is, unless it can be shown that an inconsistency is present. In practice this would be considered *verification and validation* of inconsistency identification knowledge. Claiming consistency of the reasoning knowledge is subject to the same limitations as identified for any formal models in chapter 4.

Similar arguments can be made about deterministic, rule-based approaches from the related literature (see chapter 3). However, note that, unlike in the proposed approach, such deterministic approaches limit themselves to the assumption that the pattern identifying an inconsistency is a *sufficient* condition for the inconsistency itself. That is, it is assumed that the pattern *implies* an inconsistency. Since devising a proof that the pattern logically entails the particular inconsistency is typically not given, no claim can be made about the pattern being both a sufficient and necessary condition for the state of inconsistency. Yet, the patterns are used to *classify* specific parts of models as inconsistent by looking for a very specific, fixed set of conditions. This, expectedly, leads to a potentially large number of inconsistencies that remain undetected.

7.1.3 Difficulties Associated with Identifying Inconsistencies of Semantically Overlapping Statements

A prerequisite to identifying some types of inconsistencies is the identification of a (semantic) overlap among two or more models. However, as argued throughout the dissertation, such semantic overlap is (typically) not explicitly defined, and must either be inferred or manually identified by a human. For instance, re-consider the example from chapter 4 (figure 8), where two conflicting assertions are made about the *age* of a person. One particularly grave issue in identifying a semantic equivalence in such cases is that, by nature of an inconsistency, both assertions are semantically *different* (i.e., both statements have a different semantic meaning). However, in reasoning about the inconsistency, it is assumed that the *intent* is to have semantically equivalent statements.

Identifying something that is *not* inconsistent as semantically overlapping is eased by the fact that more evidence is available in support of a semantic equivalence. However, if a semantic overlap is to be determined for something that is, in fact, inconsistent, then there is less evidence in support of the semantic equivalence and, in fact, some evidence that *opposes* this hypothesis. This complicates the problem of identifying an inconsistency in such cases, since concluding that an inconsistency is present requires sufficient evidence to *also* conclude that a semantic overlap is *intended* to be present.

Similarly, if the evidence of something being semantically equivalent outweighs the evidence of something being inconsistent, an inconsistency remains undetected. This is indicative of a trade-off that must be considered when developing reasoning knowledge. In the proposed framework, this trade-off can be accounted for quite naturally by specifying how *strongly* specific pieces of evidence influence a conclusion.

7.1.4 A Process Perspective

Ensuring that the knowledge used for reasoning about inconsistencies is free of inconsistencies and sufficiently complete (see sections 7.1.2 and 7.1.3), and is capable of producing adequate results[1], is, in most practical cases, dependent on a continuous knowledge refinement process that should be applied throughout the life-cycle of a system. Continuous refinement of the reasoning knowledge is also required by

[1]Here, *adequate results* refers to a *desirable* outcome of applying the inconsistency identification knowledge. How this can be measured concretely is discussed in the last section of this chapter.

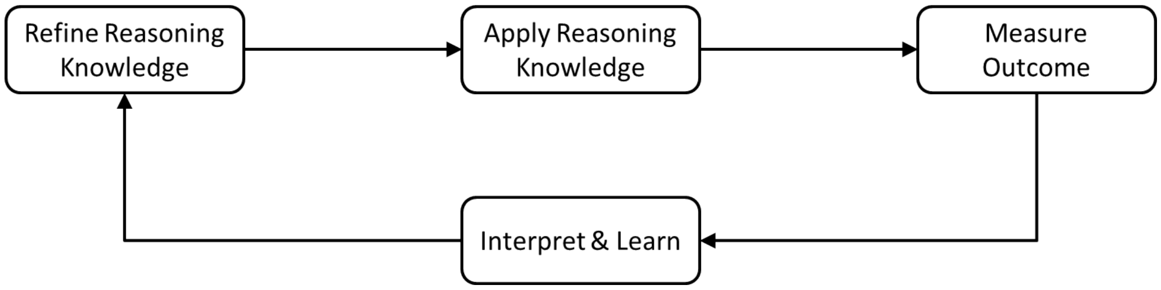


Figure 32: Life-cycle wide process encompassing continuous refinement and application of inconsistency identification knowledge. Boxes with rounded corners denote activities, and arrows between activities denote logical flows. The starting activity is intended to be the activity denoting refinement.

nature of the approach. Since the reasoning knowledge used leads to conclusions that are not necessarily logically correct, the knowledge used is often either incomplete or over-specified.

A suggestion for a supporting process to inexact probabilistic reasoning about inconsistencies is depicted in figure 32. The actual resources committed for each activity in each cycle, and the actual frequency with which this process is to be applied is a question of how much value the process adds. Therefore, it is highly dependent on the system being developed.

The process defines four activities: *refine reasoning knowledge*, *apply reasoning knowledge*, *measure outcome*, and *intepret & learn*. The application of reasoning knowledge is considered an automated process, in which models are algorithmically translated, mediated, and instances of inconsistencies identified. This is unlike the remaining three activities which *may* require human intervention.

Refining reasoning knowledge primarily entails the acquisition and elicitation of knowledge from experts or datasets. Refinement of the knowledge required for identifying inconsistencies entails both the consideration of additional knowledge and the removal of knowledge, starting from an empty collection of knowledge or an existing set of reasoning knowledge. While mostly a human process, advances in data mining allow for some automation of this activity. For instance, the structure (that is, the influence relationships) [152] and, as introduced in section 2.3.5, the parameters of a Bayesian network can be learned, provided that representative data is available (which must be considered an accurate representation of future data produced). One could even envision the process of automatically extracting patterns [33] from hand-labeled sets of inconsistent models.

Once inconsistency identification knowledge has been defined, it can be applied for the task of reasoning about possible instances of inconsistencies. As mentioned, this is considered an automated process. However, once complete, a potentially large number of propositions with uncertain truth values have been produced (see section 6.2.4). How *accurate* and *precise*, and how *complete* the inconsistency identification knowledge is should then be *measured* by analyzing these results using appropriate means. This entails, to some degree, the manual inspection of the results. The *value* of the reasoning knowledge, and the value added by refining the knowledge, should then be determined using appropriate means. In the literature, a number of measures exist for evaluating *classifiers* (such can be considered the inconsistency identification knowledge) (see, e.g., [124]). These are examined in more detail in chapter 8, where a value-based perspective is introduced also.

Finally, the results of the reasoning process should be used for purposes of *learning* – that is, the conclusions reached by the reasoning process must be *interpreted* and appropriate actions taken. This may entail explicitly *devising a strategy* used in subsequent refinement (if any) of the inconsistency identification knowledge. However, the conclusions may also go beyond the boundaries of inconsistency identification. This means that learning from the outcome of the inconsistency identification process does not necessarily entail a refinement of the reasoning knowledge. Indeed, it is possible that no refinement is conducted during the subsequent refinement stage, and that design decisions (or decisions made about the further development of the system as a whole) are impacted. If it is deemed, based on the information and knowledge available for reasoning, that the cost incurred by the inconsistencies (determined in a separate, but related process) is greater than the benefit gained by *resolving* them, the appropriate course of action may be the cancellation of the project.

In conclusion, the acquisition, elicitation, refinement and application of inconsistency identification knowledge should be treated as an integral part of the life-cycle of a system developed using a Model-Based Systems Engineering related methodology. The implications and impact on the life-cycle are very similar to those of *software testing*: an up-front investment in an infrastructure, and a commitment to continuously refining the reasoning knowledge throughout the life cycle of the system are necessary [150].

7.2 Eliciting Inconsistency Identification Knowledge

In the following, methods and suggested practices for eliciting inconsistency identification knowledge are presented. Here the focus is primarily on eliciting the structure and parameters of the Bayesian network, as well as the patterns associated with the values of the random variables.

7.2.1 Overview

When eliciting inconsistency identification knowledge, three important questions must be considered: firstly, which stakeholders are involved in eliciting inconsistency identification knowledge? Secondly, what must be elicited from whom? And thirdly, how should one elicit the required knowledge?

Who the appropriate stakeholders are for acquiring the relevant reasoning knowledge for a particular type of inconsistency depends on the type of inconsistency. For instance, recalling the classification from section 7.3.1, eliciting knowledge about identifying inconsistencies specific to a particular domain should always involve experts from the particular domain. Primarily, this includes the elicitation of random variables, their target spaces and, given sufficient understanding of the required vocabularies involved (see section 5.5), associated patterns[2].

The acquisition and elicitation of inconsistency identification knowledge should start by first defining the type(s) of entity (or entities) affected. This defines the possible outcomes of an associated experiment, where one such entity is selected at random. To identify such an outcome in the graph-based model, a *base pattern* should be defined. This base pattern builds the foundation for all other patterns, as it provides the context in which the particular type of inconsistency being reasoned about may manifest. Each type of inconsistency should have its own set of reasoning knowledge. For each such set of reasoning knowledge exactly one base pattern must be defined. Across various types of inconsistencies, this base pattern may be identical.

After defining the base pattern, direct and indirect causes for, or evidence in support or opposition of, an inconsistency and related events must be elicited. This is an iterative process which terminates once a *sufficiently complete* state has been reached. In practice, this is marked by experts deeming the further refinement of the inconsistency identification knowledge as not valuable. Within the scope of the introduced framework, eliciting such causes entails the definition of random variables and appropriate mutually exclusive, yet exhaustive target space values. That is, one must account for all possible states. The process also involves identifying influences of the causes on one another (i.e., if the probability of a particular event occurring is influenced by the occurrence of another, this should be made explicit). Once all random variables have been defined, beliefs on the network parameters must be elicited.

[2]Note that also the acquisition of knowledge about a domain for purposes of building a domain vocabulary (see section 5.5.4) should involve these stakeholders. However, building mediation rules *to* a domain vocabulary requires the collaborative effort of stakeholders familiar with the vocabulary being mediated to, who are also familiar with the source vocabulary. Note that this is not considered in detail in this chapter.

7.2.2 Defining the Experiment & Base Pattern

One of the key ingredients of any formally defined probabilistic experiment is the definition of the probability space. Recalling the definition given in section 2.3.1.1, a probability space is defined by a sample space Ω, a σ-algebra $\mathcal{F}$ and a probability measure P.

The set of possible outcomes of a probabilistic experiment is defined by the sample space Ω. An outcome $\omega_i \in \Omega$ may be a part of any number of (known) events, which are defined by the σ-algebra $\mathcal{F}$. Within the scope of the approach to inexact probabilistic inference of inconsistencies, the outcomes are understood to be the elements that inherently define the context in which a particular type of inconsistency manifests. The outcomes are also those quantities that one is interested in *measuring* in order to collect evidence.

To illustrate this, say one is interested in inferring the probability of a particular property (denoted by an edge) holding between *any pair of nodes* contained in the graph-based model. In this case, the sample space is defined as the set of all ordered pairs of nodes:

$$\Omega = V \times V = \{(v_i, v_j) \mid v_i, v_j \in V\}$$

Note that this is just one example of a sample space that may be used. However, it is representative for reasoning over graph-based models. Depending on the reasoning task at hand, Ω may be more restrictive than the formula above, or even more open. For instance, in practice, one may only be interested in pairs of nodes describing specific types of `BaseConcepts` (see section 5.5) (such as `Propertys`). In such a case, the sample space is defined by *all pairs of elements that can be identified as (base)* ***`Propertys`****.* The identification of elements that should be a part of the sample space requires answering the fundamental question:

> *"For a given type of inconsistency, what type of entities are directly*

affected by the inconsistency – i.e., what type of entities can be said to be inconsistent?"

Identifying elements in the graph-based model requires the definition of an appropriate accompanying graph pattern. This pattern must be created based on a fundamental understanding of the representation of information in the graph-based model (i.e., an understanding of how data is organized). Recall the example from chapter 6, where patient records were analyzed and the probability of lung cancer was determined based on the information available about the patient. For this example, the sample space can be defined by the set of all patients. These can be identified by the pattern (**?p**, *is_a*, *Patient*), where, for all matches to the pattern, **?p** binds to nodes in the graph-based model that represent individual *patients*. Therefore, the pattern defines the sample space, and the set of matches to the pattern represents the set of (or a subset of) the possible outcomes[3].

Figure 33 illustrates two example base patterns. The pattern illustrated by figure 33a defines the set of entities in the graph that denote `Element`s from the base vocabulary (see section 5.5.3). Figure 33b shows an example base pattern (in the form of a complex graph pattern (see section 5.3.1)) allowing the identification of all *different* pairs of entities in the graph-based model which are known to be (base) *Property*s. The functor *notEqual* ensures that only matches are returned where **?p1** and **?p2** do not point to the same node in the graph-based model.

Given that Ω is finite, the σ-algebra $\mathcal{F}$ can be defined as a subset of the power set of Ω (as was done in section 2.3.1.1). To define the probability measure, a Bayesian network is created. This encompasses the definition of a number of random variables and their respective target spaces.

[3]Note that whether the graph-based model contains all possible outcomes depends on the underlying assumptions made, which cannot be generalized.

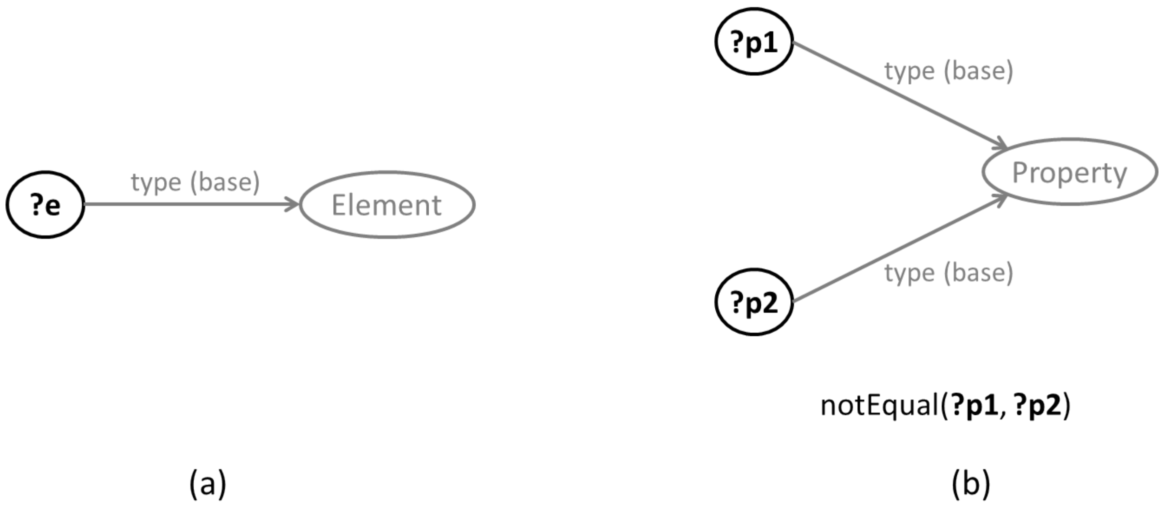


Figure 33: Example *base patterns* used for identifying possible outcomes to an experiment in a graph-based model. Note that the same coloring convention is used as in section 5.5 to differentiate statements inferred using the semantic mediation mechanism. Matching pattern (a) leads to identifying all entities in the graph-based model known to be `Element`s. Matching pattern (b) results in all pairs of *different* properties to be matched.

7.2.3 Elicitation of Inconsistency Causes & Related Reasoning Knowledge

Once it has been established what type(s) of entity (or entities) are being reasoned about and can be said to be *inconsistent*, the evidence in support (and opposition) of the particular type of inconsistency can be elicited. This includes creating the structure of the Bayesian network used in reasoning about inconsistencies, and defining patterns to be associated with random variable values.

7.2.3.1 Eliciting Elements of the Problem Domain

Eliciting the type of evidence considered *valuable* for reasoning about inconsistencies can be viewed as a two-step process: the first step encompasses determining the *possible* information that can be collected about any possible outcome, and the second step involves identifying which pieces of information (strongly) influence the probability of an inconsistency (directly or indirectly).

Determining the type of information that can be used for reasoning about the inconsistency of any outcome fundamentally requires an interpretation of the type of inconsistency and the entities involved. For instance, if the type of inconsistency being reasoned about involves the comparison of two or more entities, information and knowledge enabling this comparison must be determined (this includes information that can be extracted about the objects to be compared, and knowledge on how to compare the information). Therefore, the elicitation of reasoning knowledge requires the repeated answering of the question: *what measurable quality of any possible outcome causes this (or negatively or positively supports this)?*. This question must be answered repeatedly and, inherently, requires expert knowledge.

In the related literature, a number of methods for eliciting expert knowledge are proposed [112]. Most of these involve the gathering of knowledge either through *observation* of an existing process, answering a series of questions, or through discussion. For instance, *concept mapping* is a method of eliciting expert knowledge by generating (informal) models of domain knowledge through the definition of concepts and relationships between these. While ad hoc, many of the methods from the related literature (and, in particular, concept mapping) have empirically been shown to be effective in practice [112].

A helping aid in eliciting (relevant) inconsistency identification knowledge is an understanding of how information about an outcome (or parts thereof) is represented in the graph-based model, as well as the consideration of the statements of which an outcome may be a part. For instance, say that the space of outcomes is defined by the graph pattern depicted in figure 33b. From the definition of a (base) `Property` in section 5.5.3, it is known that elements that are types of `Property`s are related to objects that represent `Constraint`s. It is also known that a `Property` – like any other `BaseConcept` – can have a *name* depicted by the attribute `name`. Therefore, associated `Constraint`s and `name`s can be utilized as part of the reasoning knowledge.

This is part of the information that can be extracted from the *local context* of any entity depicting a (base) *Property*. Information that *indirectly* relates to the relevant property can be extracted from the larger context: for instance, since `Element`s are known to `contain` `Property`s, information about the *parent* of a property may also be extractable and used for purposes of reasoning about the respective inconsistency.

Once an initial set of reasoning knowledge has been determined, it must be completed by identifying related mutually exclusive events. In the simplest case, this leads to the identification of complementary events (if not already defined). The end goal of the elicitation process is to identify:

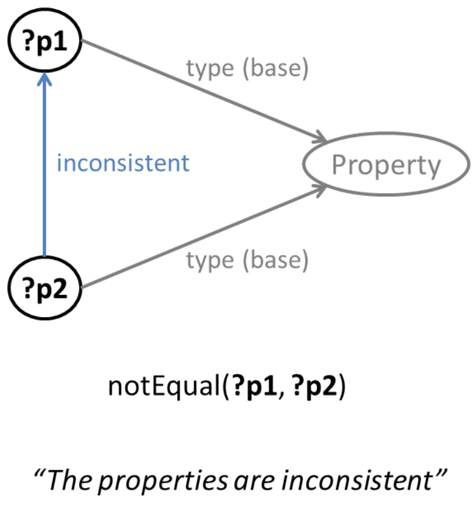
- A set of (related) random variables deemed valuable in the process of reasoning about the existence of an inconsistency
- A mutually exclusive, and exhaustive set of values for each random variable

The elicitation process may be complemented by investigating previously collected concrete manifestations of types of inconsistencies, so as to explore the context of the inconsistency and extract individual explanatory statements from it.

7.2.3.2 Defining Appropriate Patterns

Once it has been determined what information is valuable to consider when reasoning about a specific type of inconsistency, graph patterns associated with the various values of the random variables must be defined. Here, the graph patterns represent how the information *manifests* in the graph-based model. This requires knowledge about how the information is stored in the graph, a source for which is the definition of the transformations of formal models to the graph-based formalism and the various mediation rules.

Patterns should be defined for *all* random variables. This means that, in some instances, graph patterns are defined that represent how information *should* be represented in a graph (recall the example for such a case from section 6.2.4). For instance,



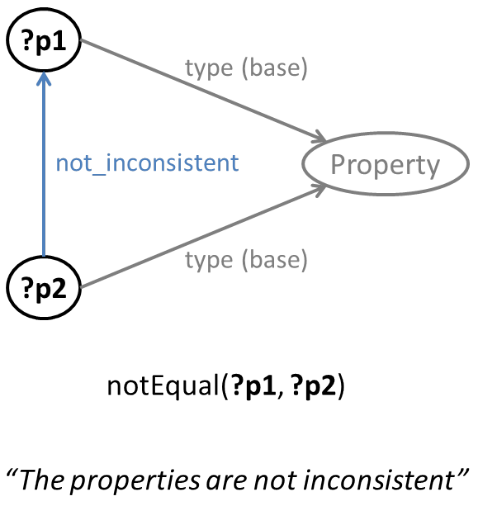


Figure 34: Graph patterns associated with the mutually exclusive random variable values *The properties are inconsistent* and *The properties are not inconsistent*. Note that by convention from chapter 5, base vocabulary elements are marked in gray, and elements from an (arbitrary) vocabulary for denoting inconsistency relationships between two entities is denoted in blue.

the graph patterns associated with the complementary events *The properties are inconsistent* and *The properties are not inconsistent* (see figure 34) are likely never matched (unless information about a particular inconsistency is explicitly stored as a *fact* in the graph-based model), but by providing a pattern, a reified instance of the pattern can be created for corresponding deductions (see section 6.2.4).

Note from figure 34 that the patterns contain the *base pattern* as a sub-graph. This is intended, since it allows for a common point of reference to be established across all patterns associated with the Bayesian network (see section 6.2.3).

An issue related to defining patterns associated with values of random variables is having to ensure that the matches to the pattern are mutually exclusive and exhaustive. That is, for every match to the base pattern, no more than one pattern associated with a random variable may refer to the same outcome. For simple complementary events (such as illustrated in figure 34) this is relatively manageable. However, more complex patterns that consider the larger context around the outcome must be crafted more carefully.

Due to the nature of the reasoning approach, validating the exhaustive and mutually exclusive nature of the patterns and values associated with random variables is challenging. Within the context of this dissertation, it is assumed that the patterns always fulfill these criteria, unless it can be demonstrated otherwise.

7.2.3.3 Independence Assumptions

Given a set of elicited random variables and associated patterns, a Bayesian network can now be constructed to make independence assumptions explicit. This encompasses the definition of causal (or influence) relationships. Causes can be defined as *"the one, such as a person, an event, or a condition, that is responsible for an action or a result"* [152]. This definition sheds light on an operational method for identifying causal relationships. That is, if the action of making a random variable take on some value from its target space (sometimes) changes the value taken on by another random variable, then the first random variable can be assumed to be responsible for (sometimes) changing the other's value. Thereby, one can conclude that the first random variable is a cause of the second. Say X_1 and X_2 are such random variables. More formally, one would say that X_1 causes X_2 if there is some manipulation of X_1 (i.e., forcing X_1 to take on some value) that leads to a change in the probability distribution of X_2 [152].

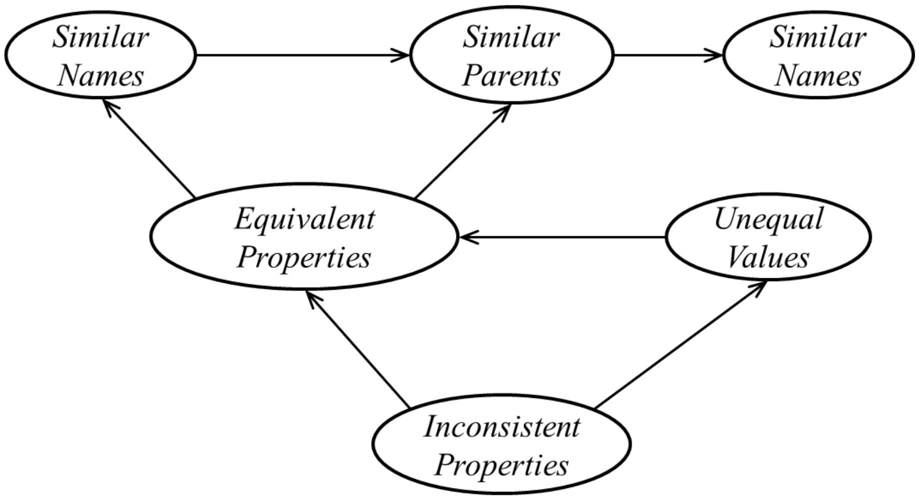
Aside from building a *general Bayesian network*, which includes all elicited influences, it is sometimes practical to make the assumption that *all evidence* for a particular event is independent. This is known as the *naïve Bayes model.* This assumption simplifies both the process of eliciting reasoning knowledge (since, by definition, all influence relationships are defined a priori), and the process of performing inference with the Bayesian network. The assumption is valid due to the *principle of maximum entropy* [18]. However, by the nature of computing with independent events, the inferred probabilities tend to be much lower than the *true* probabilities, particularly if a large number of events is considered[4]. Nonetheless, the naïve Bayes model is widely used in machine learning practice. For instance, modern spam filters, which make use of Bayesian inference to determine whether or not an incoming e-mail should be considered spam, use this model as a *basis* (often in combination with ad hoc heuristics that "soften" the effect of multiplying a large number of probabilities with one another), primarily to simplify computation and to be able to easily extend the reasoning knowledge base.

To illustrate the differences of the two models, consider the Bayesian networks in figure 35. Both are Bayesian networks designed for the task of reasoning about the inconsistency of two properties. Both also use the result of comparing the values assigned to the properties, as well as their names and the names of their parents as evidence. However, the network in figure 35a explicitly differentiates between evidence used in identifying a semantic overlap, and considers this overlap and the result of comparing the values of the properties as evidence for the properties being inconsistent. It also considers the effect of two properties as less likely to be equivalent, if their assigned values are not the same. This is not the case for the network in figure 35b, which explicitly assumes that any evidence collected from the graph-based model is independent (note that the random variables *EquivalentProperties* and *SimilarParents* were left out, by assumption of this information not being explicitly contained in the graph-based model).

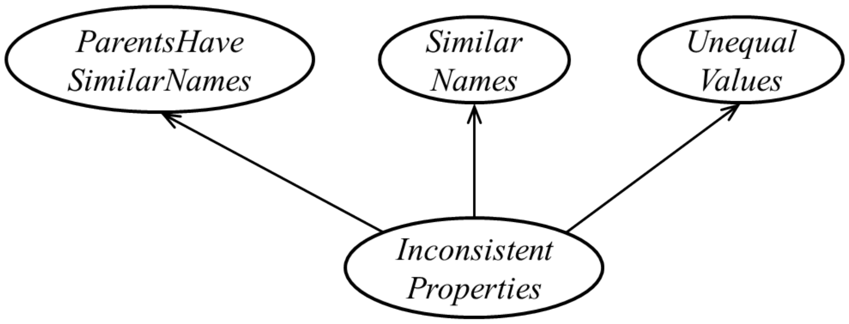
7.2.4 Elicitation of Beliefs on Network Parameters

Given a set of random variables and assumptions about their independence, one can now construct the structure of a Bayesian network. However, to fully specify the Bayesian network, the network parameters need to be specified also. As discussed in section 2.3, these parameters can be *learned* from a set of representative data

[4]This is because the joint probability of independent events is the product of the associated probabilities.



(a)



(b)

Figure 35: Illustration of a (a) full Bayesian network and a (b) Bayesian network making use of the naïve Bayes assumption. Both networks are meant to be used for the purpose of reasoning about the state of inconsistency of a randomly selected pair of properties, but in the case of (b) it is assumed that all evidence directly (or indirectly) related to two properties being inconsistent is independent.

cases. However, from a subjective Bayesian perspective it makes sense to impose a distribution which is a result of eliciting one's belief on the value of the network parameter. These distributions can then be updated with data, if desired. As discussed in section 2.3.5, a convenient parametric distribution to use for capturing beliefs on Bayesian network parameters is the Dirichlet distribution.

Discrete probabilities can be elicited by considering *one's willingness to bet* on an event. Specifically, the elicitation of subjective beliefs entail asking the following question:

> *"How much would you be willing to pay for a gamble in which you earn* \$1 *if the (given) event occurs and* \$0 *if it does not?"*

Let $\$b$ be the amount one is willing to pay. The subjective probability can then be expressed as $p = \$b/\$1$. Note that one must be willing to both buy and sell the bet – that is, it must be a fair price for the bet.

Eliciting beliefs in the manner described above has its caveats. Primarily, a number of biases influences the selection of a fair price. Examples of such biases are *motivational bias*, where the true believe may not be expressed due to there being an incentive to bias the distribution in a particular direction, and a bias with respect to availability of information, where the assessment of the probability that an event will occur is linked to the ease with which a stakeholder can remember similar events [217, 126, 95]. Additionally, if the occurrence of the particular event being considered is very low (or, similarly, very high) (i.e., $P(E) < 0.01$ or $P(E) > 0.99$), determining a probability that is close to the true value and not off by one or more orders of magnitude – which can have a significant impact on the inferences – is difficult, since humans are not very good at quantifying such probabilities. Indeed, humans tend to generally underweight outcomes [126, 217]. Therefore, it can be valuable to update the probability distributions on such network parameters by learning from data that is incrementally collected, so that any bias is mitigated. However, as discussed in section 2.3.5 the amount of data required for this is typically very large.

To exemplify the introduced concepts and concerns, consider, once more, the Bayesian networks from figure 35. The distribution associated with one of the network parameters is defined by $p(InconsistentProperties)$. Assuming that there are only two possible states – *inconsistent*, and *not inconsistent* – it suffices to ask the above question for one of the two possible events. That is, *how much would you be willing to pay for a gamble in which you earn* \$1 *if a pair of properties that is randomly selected from a given sample space is inconsistent, and* \$0 *if it is not inconsistent?*. For a randomly selected pair of properties, this probability is likely going to be very small and difficult to assess correctly.

One possible practical way of mitigating issues related to eliciting beliefs on highly infrequent events is to modify the Bayesian network in such a way that the associated network parameters are easier to elicit. For instance, it may be much easier to assess the probability of a randomly selected pair of properties being inconsistent *given the knowledge that the properties are also semantically equivalent.*

7.2.5 Reuseability of Inconsistency Identification Knowledge

Particularly in model-based development scenarios where a large number of heterogeneous models are used, or the cost invoked by undetected inconsistencies is very high, the cost incurred by eliciting inconsistency identification knowledge is not negligible. Therefore, strategies should be put in place that reduce this cost. One aspect which, arguably, has a high impact on this, is the reuse of previously defined inconsistency identification knowledge.

Given that some types of inconsistencies are independent of a domain, application or use-specific context (see section 7.3.1), it is conceivable that at least a part of the knowledge elicited for the purpose of identifying such inconsistencies (e.g., the

Bayesian network structure and associated patterns) can be reused across various system developments. However, whether *all* knowledge can be reused depends on the expected boundary conditions. Of particular concern are the network parameters: since the network parameters encode believed frequencies of certain events (such as the expected degree of inconsistency), the same (or at least highly similar) frequencies must be expected in the scenario where the knowledge is reused. That is, the belief on the network parameters must remain unchanged. However, it can be argued that this is unlikely the case in practice (at least in general) since it assumes no learning effect.

Reuse of patterns is enabled by the semantic abstraction mechanism (see section 5.5). For maximum reuse across different development scenarios, patterns should not be overly specific, and not too broad. That is, they should (ideally) represent necessary and sufficient conditions for the property of the outcome that they imply. Refinement of the knowledge over the course of the life cycle intends to support this. Generally, patterns using language-independent vocabularies can be said to be more reusable across scenarios, since they do not depend on possible frequent changes to language specifications, and only need to be updated if the domain or base vocabulary changes (which is assumed infrequent).

7.3 Interpreting & Presenting Inference Results

By nature of the suggested reasoning approach (see chapter 6), the number of inferred propositions (with uncertain truth values) can be very large. For instance, when reasoning about the inconsistency of pairs of properties, and using a base pattern similar to the base pattern from figure 33b, a *pairwise* comparison is performed. That is, (without the *notEquals* functor) n^2 comparisons are performed for n unique properties. Even for a relatively modest number of properties distributed across the various models (say $n = 500$), a very large number of possible inconsistencies (with non-zero probability) are inferred (for $n = 500$ this would result in $250^2 = 250,000$ deductions).

Note that only a small fraction is expectedly relevant; the probability that *any* randomly selected pair of properties is inconsistent is, arguably, very small. By nature of the approach, it is vital that possible inconsistencies are investigated by a human (who is assumed to be able to identify whether or not an instance of a particular inconsistency is present with perfect accuracy) (or some other *oracle*) since the inferred statements cannot be claimed to be logically correct (even if the determined probability is very high) (see chapter 6). However, manually checking every deduction is very costly[5]. Therefore, it is prudent to investigate whether it is possible to *restrict* the set of possible inconsistencies presented to a human for confirmation and, if so, by what means. This is discussed in the following.

7.3.1 Classification Heuristics

A commonly used method for inexact reasoning approaches is the use of a *cutoff* value for deciding whether or not it is valuable to present a result to a user. This can be considered a *classification heuristic* (or *decision heuristic*) that is used for deciding which class a particular outcome belongs to (i.e., here: *inconsistent* or *not inconsistent*). For binary classifiers, its implementation is fairly simple: given a cutoff value c, the event with a probability higher than or equal to the cutoff value is considered to be `true` (i.e., `IF` $P(Event) \geq c$ `THEN` $Event$ `ELSE` $\neg Event$).

While the use of a cutoff value can considerably reduce the number of statements (by *"cutting off"* cases with low probability), choosing a sensible value for the cutoff is non-trivial. Choosing a value that is too low results in a potentially large number of results being presented to a user that were wrongly classified as inconsistencies. On the other hand, choosing a value that is too high may result in *not* detecting some

[5]This claim is investigated in chapter 8.

actual inconsistencies. The selection of a cutoff value also depends on the expected cost incurred by *wrong classifications* and *missed inconsistencies.* Therefore, the *value* (or utility, or benefit gained by) of some cutoff values is higher than that of others. By nature of inconsistencies, it is expected that the cost incurred by an actual inconsistency that remains undetected due to *not* being a part of the set of cases investigated by a human is much higher than the cost incurred by being exposed to a wrongly classified case.

An alternative to using a cutoff value is the heuristic of always choosing the MAP event (see section 2.3.2). That is, the event with the highest probability is chosen for each outcome. This is a common approach in machine learning, but is known to, in general, only produce sensible results if the true frequency of the events is at least similar [149].

7.3.2 Presenting Inference Results

Since all inferred propositions have a probability associated with them, it is sensible to *rank order* the results in descending probability. Results with higher probabilities can be argued to have been derived on the basis of more (or more strongly) supporting evidence. Similarly, the derived statements may be *clustered* by grouping them by probability ranges (e.g., statements with probability 0.8 to 1.0 may be grouped under cases that are to be investigated with high priority). Within a group, rank ordering may be utilized as well.

While rank ordering and clustering are sensible options to decrease the complexity and limit the cost associated with having to confirm instances of inconsistencies, it is not always an effective means of presenting only the most relevant results. For instance, if a large number wrong conclusions are presented to a user, this may be an indicator that *refinement* of either the structure (and / or patterns), or the parameters of the Bayesian network is necessary.

7.3.3 Learning Network Parameters & Refining Reasoning Knowledge

Since, by nature of the approach, the conclusions reached about an instance of an inconsistency being present cannot be claimed to be logically correct, humans (who are assumed to be able to identify inconsistencies with perfect precision), or some other mechanism capable of perfectly identifying inconsistencies (similar to what would be considered an *oracle* in software testing [150]), must review at a minimum the subset of the inferred statements deemed the most relevant or most valuable to investigate further. This investigation of individual results potentially reveals more information about a particular outcome. Therefore, it is sensible to use what has been learned from the process of investigating the particular cases to improve future classifications. In particular, the results can be used for updating the distributions on the network parameters, which can be done using the methods introduced in section 2.3.5.

It should be noted that primarily using instances of one class (especially if these instances are infrequent among the set of possible outcomes), the network parameters may be biased towards certain cases. Ideally, training should occur by uniformly sampling from the set of inferred statements to conserve a consistent sample size across the Bayesian network [152]. However, as argued before, such an approach requires a very large number of cases to be considered to improve classification accuracy.

7.4 Summary

In this chapter, implications of inexact reasoning about inconsistencies in sets of heterogeneous models using the proposed method from chapter 6 are discussed. The first part of the chapter introduces the notion of *inconsistency identification knowledge* which, per the given definition, consists primarily of a description of the class of the entity (or entities) associated with an instance of a particular type of inconsistency (i.e., the definition of the *outcome* of an experiment), the definition of the structure and parameters of a Bayesian network, and the patterns associated with the values of the random variables in the Bayesian network. In the second part of the chapter, the elicitation of this inconsistency identification knowledge is discussed. Finally, the third part discusses important aspects of interpreting and presenting inference results.

An important insight from the first part of this chapter is the problem of having to identify an *intended* semantic overlap as part of identifying an inconsistency. Detecting such intended overlap is non-trivial since an inconsistency can lead to very different semantic interpretations of the involved entities. The elicited reasoning knowledge must account for this. A conclusion drawn from the discussion is that, in practice, this may lead to a (necessary) reduction of precision of the approach, since weaker assumptions must be made when measuring similarity. A second important insight is that inconsistency identification knowledge should be refined over the course of the life-cycle to improve both accuracy and precision.

The second part of the chapter is concerned with the elicitation of knowledge. A number of methods are suggested for eliciting both the structure and parameters of the Bayesian network, as well as methods for eliciting patterns. Important is the insight that the elicitation of the network parameters is non trivial due to the possibility of some events associated with inconsistency identification being expectedly rare (e.g., the probability of any pair of properties being inconsistent is very small). The non-triviality stems from the empirical observation by researchers that humans are not very good at quantifying the probability of such rare events: humans tend to generally underweight outcomes, which is representative of one of several identified biases. Because the elicitation of inconsistency identification knowledge is expected to be high, aspects of knowledge reuse are briefly discussed.

In the last part of the chapter, the interpretation and presentation of inferred probable inconsistencies is discussed. It is acknowledged that the set of possible inconsistencies (and semantic relationships) is, in general, very large. A need is identified for preparing the results in such a manner that they can be inspected more easily. Proposed are rank ordering (according to (posterior) probability) and clustering, as well as the use of decision (or *classification*) heuristics.

CHAPTER VIII

EVALUATION & CASE STUDY

In this chapter, the inexact probabilistic reasoning approach introduced in chapter 6 is applied to the case of inconsistency identification, and its characteristics within this context are investigated. Specifically, the expected behavior discussed in chapter 7 is analyzed. Factoring in the evidence presented in section 4.1, a global *validation* is, as discussed in section 1.4, impossible. Therefore, both a qualitative and a quantitative analysis is conducted to collect as much evidence in support (and opposition) of the effectiveness and efficiency of the approach.

The primary aim of this chapter is to provide both qualitative and quantitative results for assessing the overall effectiveness and efficiency of the approach. The results act as evidence to assess the validity of hypotheses 1 through 5. A qualitative evaluation is done by examining the theoretical computational complexity of the underlying algorithms. The approach itself is evaluated quantitatively by measuring the performance of the classifier. For this purpose, a prototypical implementation of the underlying algorithms and a supporting infrastructure has been developed. To measure the performance, several measures and metrics are defined, through which the approach is evaluated and comparative analysis performed.

The chapter is structured as follows: in section 8.1, the supporting tool infrastructure is introduced briefly. Thereafter, the utilized case study and results of the quantitative evaluation are presented in section 8.2. Finally, in section 8.3 the complexity of the underlying algorithms is analyzed both theoretically and empirically.

8.1 Proof-of-Concept Tool Support

A *semantic web* [19] inspired approach was selected as a basis for implementing the concepts developed in chapters 5 and 6. The use of a web-based approach as a basis for the implementation of an inconsistency identification tool-suite is advantageous, in that an environment is used that was designed – from the start – for the purpose of storing, linking and retrieving information and knowledge from distributed sources. This is useful, since, in practice, models are typically scattered across various repositories and on various physical machines. The core focus of the *semantic* web is the *"integration / combination of data from diverse sources, whereas original Web concentrated on interchange of documents"* [19]. This is an added benefit, since the integration and combination of data subsumes interpretation – hence, *semantic* web. An initial version of the proof-of-concept tool infrastructure is published in [108].

8.1.1 Leveraging RDF as a Concrete Implementation of a Common Graph-Based Formalism

For semantic web applications, the *World-Wide Web Consortium* (W3C) recommends the use of the *Resource Description Framework* (RDF) as a knowledge representation method. RDF is compatible with the representational aspects of the concepts introduced in chapter 5 and can act as a concrete implementation of a common representational formalism. In part, this is due to (a convenient mental model for) RDF being *graph-based* and allowing for individual propositions to be expressed as *subject-predicate-object* triples. RDF 1.1 makes use of *Internationalized Resource Identifiers* (IRIs) (a subset of which are *Uniform Resource Locators* (URLs)) as a means of uniquely identifying *resources* (subjects, predicates or objects).

RDF defines several *Classes* and *Properties* (see [227] for a full listing). The semantics of RDF are intentionally very weak, with only a few structural constraints being enforced (e.g., datatype violations). RDF defines a simple typing mechanism, in which *type-of* (instance / individual - of) relations are defined. Furthermore, specialization (subclass) relationships are defined, which are, by the RDF semantics, transitive.

8.1.2 Logical Inference Engine & Query Processor

The developed prototypical tool support builds on the open source RDF handling framework *Apache Jena*[1] (henceforth only referred to as Jena). Jena is divided into a variety of components: one large part of the Jena framework is a rule-based reasoning system, which supports a syntax similar to standard *datalog* implementations as a rule formalism. Supported are both forward and backward deductions. The implementation of the inference mechanism is appropriate for the given context, since it allows for the definition of *functors* (similar to those that are introduced as part of complex patterns (see section 5.3.1)). Jena's generic rule reasoner is used for implementing the semantic abstraction mechanism. A second major component of Jena is ARQ, which is an implementation of a SPARQL 1.1 [226] compliant RDF query processor.

8.1.3 Bayesian Reasoning Engine

As a basis for the Bayesian reasoning engine, a library supporting the construction of, and performing inference in Bayesian networks (named `JBayNeT`) was developed. The library supports discrete random variables and various inference methods: naïve enumeration, variable elimination, and the junction tree algorithm. Bayesian networks can be exported in the *Bayesian Network Interchange Format* (BIF) version 0.3 format, which is supported by most commercial and open source Bayesian network tools. The library also supports learning of general Bayesian networks. Note that while open source implementations of Bayesian network libraries exist, none were found to support all required features[2].

[1]http://jena.apache.org

[2]The source code for this library or the Bayesian reasoner has not been released at the time of writing this dissertation, but there are plans of doing so in the future.

A large part of the proof-of-concept implementation is also the implementation of a reasoning engine based on the algorithms presented in chapter 6. For this purpose, a reasoner implementing Jena's `Reasoner` interface was built. The `Reasoner` interface is an interface used by Jena's forward- and backward-chaining rule reasoner implementations. By ensuring compatibility with Jena's reasoning framework, the expressiveness for defining patterns using the Jena Datalog-like rule language is preserved by internally rewriting the patterns used for measuring random variables as rules with empty rule headers. This allows for implementation reuse of Jena's pattern matching algorithms.

8.2 Quantitative Approach Evaluation

In the following, the presented Bayesian approach to inconsistency identification is evaluated quantitatively. This is done in order to assess characteristics (such as effectiveness) and measure qualities of the approach which cannot be done by theoretical evaluation of the underlying algorithm alone. As part of the quantitative evaluation, the influence of the size of the evidence set considered, as well as the effect of learning is investigated.

The evaluation is based on the proof-of-concept implementations of supporting software tools introduced in the previous section. Additional software has been written for the purpose of creating a controlled environment, collecting data points and computing evaluation metrics.

8.2.1 Overview & Evaluation Goals

The overall goal of the quantitative evaluation is to investigate the validity of the proposed hypotheses of this research. Of primary interest is investigating whether the approach is capable of identifying those inconsistencies and associated semantic overlap that cannot be effectively entailed by deterministic rules. This includes testing the ability to identify inconsistencies and semantic overlap under conditions of incompleteness and under the presence of inconsistent information. However, also included is the evaluation of how much knowledge must be encoded and elicited in order to identify certain kinds of inconsistencies and semantic overlap, as well as the value of the probabilistic deductions. In particular, the latter allows for an assessment of how much human involvement is required, and whether the approach can be economical. Lastly, a goal of the quantitative evaluation is to identify current limitations and potential for future research.

Assessing the characteristics and making assertions about properties of the presented approach requires a large number of data points. For this purpose, a number of sets of related (and overlapping) heterogeneous models of *Railway Systems* are generated and injected with inconsistencies, incompletenesses and other random errors that are intended to simulate concurrent evolution of the models (and human error / preferences). The information and knowledge encoded in these models is then mediated to the abstract vocabulary of the base ontology presented in section 5.5. Reasoning under different assumptions and using different Bayesian networks is then performed. During this process, a number of measurements are taken which are used in evaluating the hypotheses of this research.

Using only expressions from the base ontology, two Bayesian networks and associated patterns are created, each of which is independently used in reasoning about inconsistencies and semantic equivalence of pairs of properties. Both of the Bayesian networks predict the probability of semantic equivalence and inconsistency, but consider a different set of evidence and context. While the one network uses information that is mostly syntactical in nature, the other uses semantic information from the larger context around the properties (such as information about the entities owning the properties). This is done to assess the amount of knowledge required to be encoded to effectively identify inconsistencies and semantic equivalences. In a second set of tests, the prior beliefs on the parameters of the Bayesian networks are updated incrementally by informing the network with instances of actual semantic equivalences and inconsistencies. In a third set, this experiment is repeated, but a set of uninformed beliefs on the Bayesian network parameters is assumed. These tests are performed to assess the impact of learning, and whether an improvement can be observed over using subjectively elicited beliefs.

8.2.2 Case Study: Analysis of Heterogeneous Models of Railway Systems

In the following, the structure and nature of the various randomly generated models of railway systems are introduced. Details about the nature of the introduced semantic overlap, as well as the types of inconsistencies and incompletenesses that are randomly injected, are also given.

8.2.2.1 Scenario

Railway systems are an important part of the public transportation infrastructure of most major cities and countries. While railroad tracks are (at least in some countries) owned by the state or some other external entity, privately owned companies typically lease, and are responsible for serving particular *routes* that are part of a *railway system*. A route can be defined as a series of *segments* between various stations, the length of which is variable. In addition to track segments, *switches* are a typical part of a network of track elements, which branch off other routes. The *positions of switches* determine which route is actively being served. In practice, routes are often marked by *signals* (which have different states) at the respective start- and end-points.

Various concerns must be addressed when designing railway systems. These concerns are both technical and economical in nature, making it a suitable case study for (model-based) systems engineering: for instance, the cost of maintaining a route is an important consideration in determining the fare price. However, this is also influenced by the size of the railway network and length of potential routes, which may only be

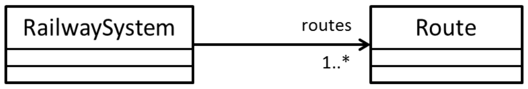


Figure 36: Meta-model used in generating *system models* for railway system designs, in standard class diagram notation (adapted from [218]).

available from an external stakeholder (i.e., the owner of the railway track network). Other concerns, primarily of a technical nature, include the placement of signals and positions of switches for serving various routes. While the former concerns require only little detail about the route, addressing the latter makes a detailed description of the route and the various components placed along it inevitable.

To illustrate and simulate a scenario in which models of different granularity and levels of abstractions are used simultaneously in a system design process, three distinct types of models are considered: *system models* to describe the railway system and the routes from an abstract point of view, detailed *route models* to outline technical details of a single route that is a part of the railway system, and *track network models* to describe the infrastructure available for planning routes. It is assumed that models of all three types are developed by different stakeholders and using different modeling languages. Inconsistencies in the models are to be identified.

8.2.2.2 Meta-Models

Meta-models for all three types – system models, detailed railway route models, and track network models – are depicted in figures 36 – 38 in standard class diagram notation. These meta-models are adapted from the domain-specific modeling language for railway routes developed as part of the MOGENTES EU FP7 project, which was also used by Ujhelyi *et al.* in testing the performance of retrieving information from large model instances within the IncQuery framework [218]. The meta-models are designed in such a way that they contain most constructs typically used in class diagrams. Railway related concepts, relationships and constraints on the syntactical well-formedness of the models were sourced from experts from the railway domain.

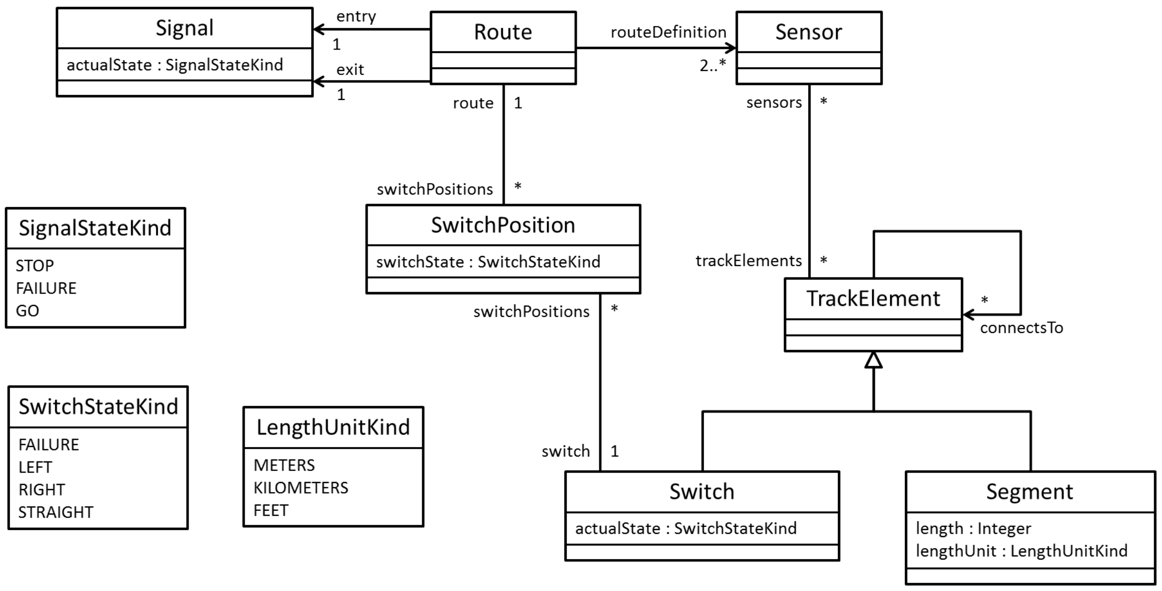


Figure 37: Meta-model used in generating detailed railway route descriptions, in standard class diagram notation (adapted from [218]).

The *system model* is used in describing the various *routes* that are a part of a particular *railway system*. It is assumed that a railway system must necessarily have at least one route defined. For this purpose, the meta-model of the system model defines two classes: `Route` and `RailwaySystem`. The class `RailwaySystem` has one attribute named `routes`, which signifies its relationship to the class `Route` through a directed association relationship with cardinality `1..*` (i.e., *one or more of*). Therefore, a railway system model is well-formed iff it defines at least one instance of `RailwaySystem` and at least one instance of `Route`, where all instances of `Route` are related to exactly one instance of `RailwaySystem`. The meta-model for railway system models is depicted in figure 36.

Detailed route models are used in refining the description of railway routes. Each route model describes exactly one route. Routes have *signals*, which mark their entry and exit point. Signals either allow trains to pass through (state *go*), request trains to stop (state *stop*), or require maintenance (state *failure*). Routes are defined by a series of *sensors*, which are installed next to *track elements*. These track elements are

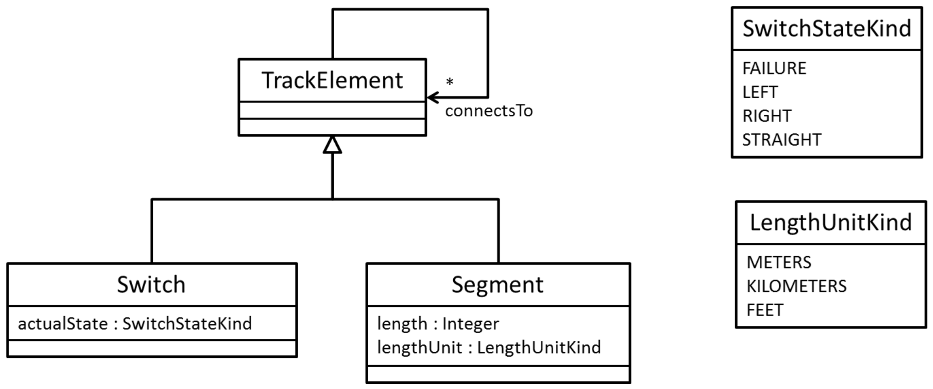


Figure 38: Meta-model used in generating models of railroad track networks, in standard class diagram notation (adapted from [218]).

connected to one another and can be either track segments or switches. Segments have a defined length and accompanying unit (of meters, kilometers or feet). Switches mark points at which other routes may branch off. Switches can be in one of four states, defining their set direction (left, straight, right) and status (failure). In a larger network of tracks, the segments and positions of switches define a route.

The meta-model used in creating (detailed) models of routes is depicted in figure 37 and defines eleven classes, three of which are enumeration kinds: `SignalStateKind` is used in describing the state in which a `Switch` can be, `SwitchStateKind` to describe the state of a `Switch`, and `LengthUnitKind` to specify the length unit when defining the length of a `Segment`. `Segment`s and `Switch`es are special kinds of `TrackElement`s, both of which may be connected to other `TrackElement`s. In order for a detailed route model to be well-formed, the model must be a valid instance of the meta-model, and meet the following additional constraints: all instances of class `Switch` must be related to at least one instance of class *Sensor*, and the length attribute of each `Segment` instance must be greater than zero. It is assumed that a length unit must not be specified.

Finally, to model railway track networks, the meta-model depicted in figure 38 is used. Similar to the models describing routes, the meta-model for track models defines classes for `TrackElement`s, which may be connected to other `TrackElement`s. These

`TrackElement`s may be either `Segments` or `Switch`es. Unlike the models of routes, no sensors or signals are defined. However, a track network model is assumed to define more than just those segments and switches found in detailed models of routes – it is meant as a model of a railway infrastructure irrespective of routes defined in any other models. Railway track network models are interesting to study not only because of their overlap with the other models describing the railway system, but also due to the fact that track networks typically contain cycles. Such cyclical models provide an interesting case for testing and evaluating the performance of reasoning algorithms.

It should be noted that the meta-models are not comprehensive enough (and, hence, the models derived from these are not expressive enough) for commercial applications. However, they are sufficiently complex for evaluating the approach. This can be justified by the following: firstly, most of the commonly used elements of class diagrams are employed (as mentioned in section 2.1, these are the de-facto standard for defining meta-models of modeling languages). Secondly, while the meta-models overlap partially, there is no obvious 1-to-1 correspondence between their instances. The latter makes it particularly hard for a model-to-model transformation mechanism to be employed for synchronization purposes, unless all possible instance-level correspondences are known a priori (which is assumed to *not* be the case). Thirdly, because certain attributes (such as the length unit) are optional, the models cannot be assumed to be complete. Therefore, a semantic interpretation of the models is inevitable.

8.2.2.3 Generation of Random Instances

Model instances are generated algorithmically based on the meta-models introduced in the previous section. For this purpose, the *Eclipse Modeling Framework* (EMF) is used, and the respective meta-classes generatively transformed to Java classes. An algorithm was implemented that allows for the automated generation of instances of

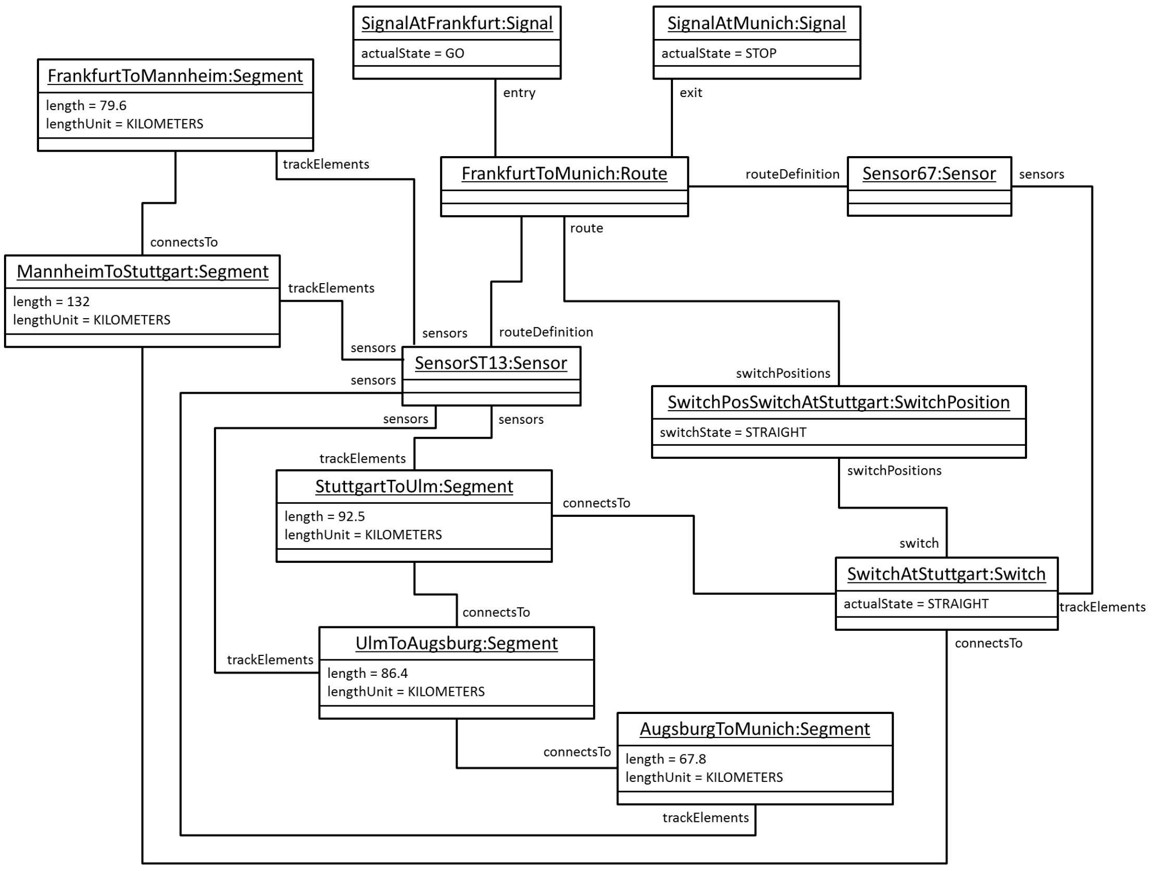


Figure 39: Sample instance of the detailed railway route description meta-model.

these models. During this process, all known semantic equivalences are stored in a list, but are *not* made accessible to the Bayesian reasoning algorithm.

The first step in generating the models is the generation of a railway system model, which encompasses generating one instance of class `RailwaySystem`, and n_r instances of class `Route`. Next, n_r models detailing each of the routes defined in the system model are generated. This is done by first generating two instances of class `Signal`, which mark the entry and exit points of the route. n_{sen} instances of class `Sensor` are then generated. For each of these sensors, $n_{seg}+n_{sw}$ track elements are added, at least one of which is an instance of class `Switch`, the rest being instances of class `Segment`. For each instance of class `Switch`, a linked instance of class `SwitchPosition` is created as well. Figure 39 shows an example of one detailed route model instance, illustrating a typical route served by the German railway system from *Frankfurt* to *Munich*.

In a third step, a railway network of a random size is generated. This network contains instances of segments and switches that are semantically equivalent to those created in the process of generating route models, and features additional track elements due to branching off at the switches. The number of track elements is chosen at random. Note that in the algorithmic implementation, the routes are generated in such a fashion that the entry point of one route is the exit point of the previously created one, looping the exit point of the last route back to the first route, thereby creating a ring of routes. Note that the fact that the first and last segments of different routes may be connected is not represented in the detailed route models, but only in the track network model.

To give the instances of each class sensible names, Princeton University's WordNet® [171] database is polled for random sets of nouns. Free for academic use, WordNet® is a large lexical database of English nouns, verbs, adjectives and adverbs, all of which are grouped into sets of cognitive synonyms (so called *synsets*), each expressive of a distinct concept. Synsets are interlinked by means of conceptual semantic and lexical relations.

Each of the random nouns extracted from the WordNet® database are intended to reflect a particular *location* in a railway network. Routes, signals, segments and switches are given names that reflect the noun chosen for the particular location. For instance, the names of the entry and exit signals each contain words that reappear in the name of the route. The first and last track element of a route also contain these nouns. Names given to switches reflect the single destination to which all connected segments lead, or from which they branch off. This naming scheme is exemplified in figure 39 with names of cities in Germany. Values to attribute slots are also provided according to a specific schema: the *length* of each segment is determined by choosing a random number between 1 and 1000 (from a uniform distribution of integers), with the unit of length being `LengthUnitKind.KILOMETERS` in all instances. A value for the slot indicating the state of a switch is chosen at random.

Finally, unique identifiers for each element are created and each element is associated with a particular model. The unique identifiers are simply a random number. The association with a particular model is done based on a generated name of a model. Model names are unique for a system model, for each route model, and for the railway track network model, and are generated by a similar pattern. For instance, the name of a railway track network model is prefixed with *"RailwayNetwork_"*, then concatenated with a randomized form (see next section) of the name of the railway system. This string is then appended with one or all of the following (and, at random, synonymous variations of these): with a probability of 0.3, the word *"-v"* followed by a random number between 0 and 9 signifying the version of the model, and (at random, with probability 0.5) either the word *"original"*, *"final"*, *"draft"*, or *"forReview"*. A *domain name* is also generated for each model, signifying a part of a *resource location* (such as a folder, drive, or network location). The name generation for each model is designed to mimic a typical naming scheme observed in practice.

The number and size of models describing railway systems can be varied by altering the overall number of routes (n_r), the number of sensors (n_{sen}), the number of segments (n_{seg}) and the number of switches (n_{sw}). This enables the collection of data from models that have similar characteristics (determined by the algorithmic construction), but are different in size and, to some extent, topology.

8.2.2.4 Semantic Overlap of Models & Randomly Injected Inconsistencies and Incompletenesses

Both the meta-models and their generated instances overlap semantically. This overlap is illustrated in figures 40 and 41.

As mentioned in section 8.2.2.2, both a *system model* instance and the various instances of the *route models* allow for specific *routes* to be defined. Therefore, both models allow for individuals belonging to the class of *railway routes* to be created.

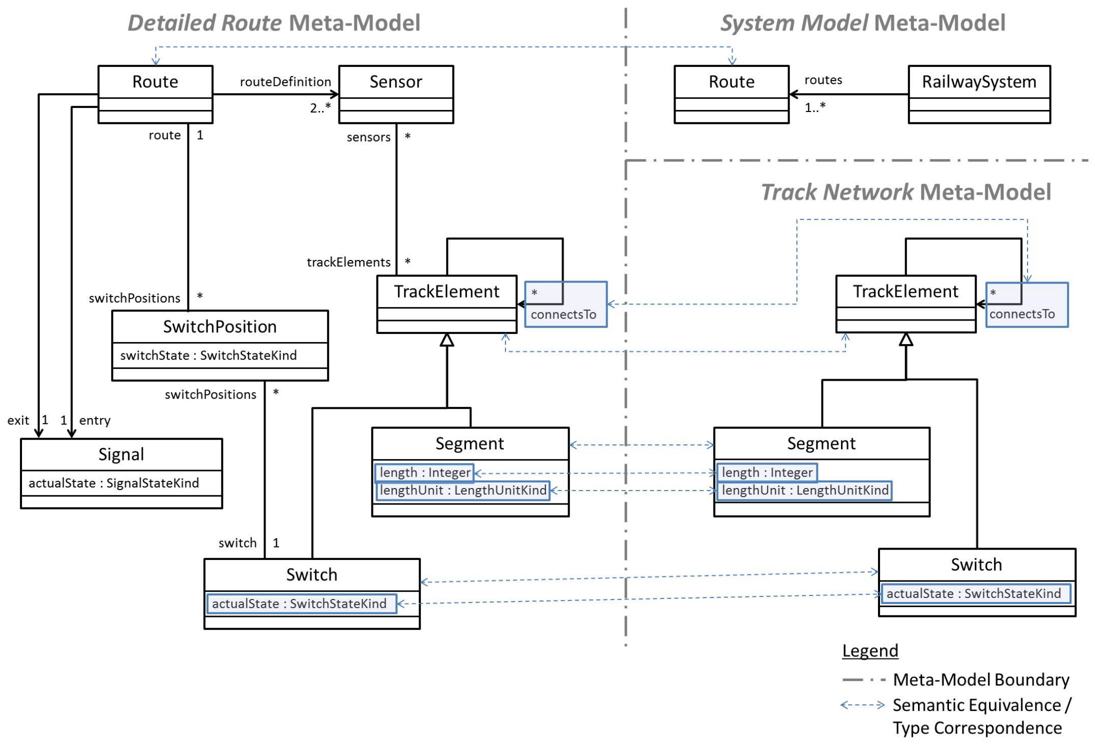


Figure 40: Type-level semantic overlap (correspondences between meta-models) across the different meta-models used. Note that correspondences between enumeration classes are left out for brevity, but are analogously defined.

For this purpose, both of these meta-models incorporate syntactic elements whose meaning is defined by the semantic concept of a *route*. However, no relations to other concepts from the railway domain (or other related domains) are shared by the meta-models. Similarly, the concept of *track elements* – and specifically *segments* and *switches* – are a part of both the meta-model for defining detailed models of routes and of the meta-model used in generating models of railway track networks. Dissimilar to the previous case, these meta-models do share some attributes and relations to other concepts from the railway domain: a length and accompanying unit is defined for segments in both cases, and switches are given a *"state"* attribute. The meta-models also share the ontological relation *connectsTo*, which represents the relationship between individual track elements. However, relations such as those from individual track elements to individual sensors are not shared by the track network meta-model. These *meta-model correspondences* are summarized in figure 40.

Note that the meta-model correspondences are identified based on an assumed semantic mapping from meta-model elements to elements from an assumed universally-accepted, well-understood, and likely finite railway domain model. This domain model contains all concepts and their relations that are relevant to the railway domain (for practical purposes this can be thought of as an ontology). The semantic interpretation of the meta-model elements – and, hence, the *types* of things representable in the language – is relatively straightforward, given a formal representation of the railway domain (refer to section 2.1.1.4 for a discussion on semantic domains and the necessity of a syntactic representation of these). However, the semantic interpretation of the possible *utterances* of the respective meta-models requires a different semantic domain. For instance, while one can entail that the object `FrankfurtToMunich:Route` in figure 39 is *a kind of* (or *of type*) *Route* (for which the existence of a semantic mapping is assumed), one cannot establish the semantic difference (or equivalence) to other routes – say, `DüsseldorfToFrankfurt:Route` –

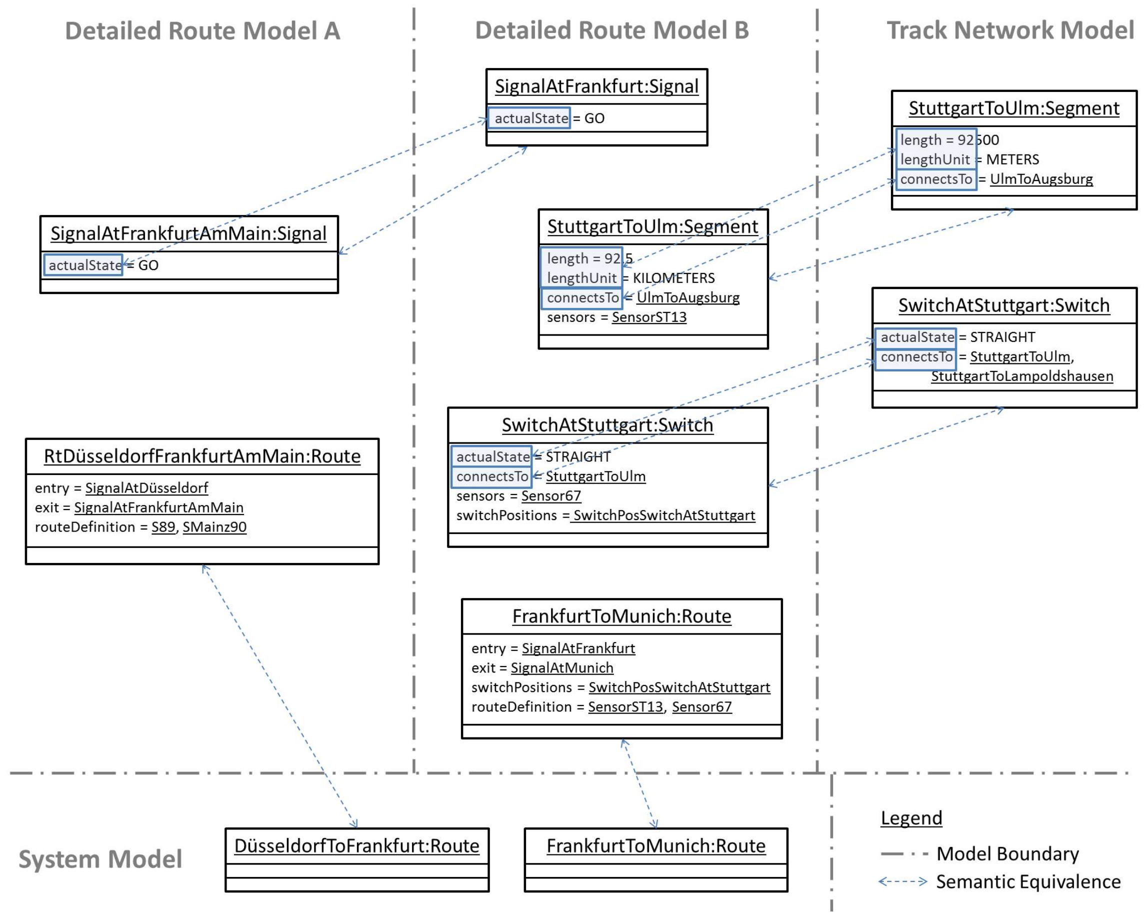


Figure 41: Instance-level semantic overlap (*"same thing"*) across system model, route models, and track network model. Note that type-level overlap (*"same kind of thing"*) is not shown (see figure 40 for an illustration of these).

without an appropriate definition of a semantic domain that distinguishes the two (to a human sometimes "obviously") different routes. On the other hand, defining such a semantic domain is non-trivial – if at all possible – due to the great number of possible instances that can be created. This complicates the problem of identifying a semantic overlap of such instance-level models.

Figure 41 illustrates the nature of the semantic overlap between system models, detailed route models and track network models. Note the use of different units and values (e.g., *92.5 kilometers* and *92500 meters*) and the synonymous expressions used (e.g., *FrankfurtAmMain* instead of *Frankfurt*). Even though syntactically different, some expressions are semantically equivalent. As established in section 2.1.1.4 already, this can only be determined with certainty given an appropriate interpretation function. The challenge for the Bayesian approach to inconsistency identification is to identify this overlap and any possibly inconsistent expressions.

To simulate situations similar to those illustrated in figure 41, a combination of one or more variations of spellings, synonyms, spelling mistakes, and variations of naming conventions are integrated at random. In addition, prefixes and postfixes such as random numbers or appropriate syntactic expressions (e.g., "RT" for routes) are added at random. For instance, the name of a route from *Frankfurt* to *Munich* may become *FrankfurtToMunichRoute* by default during the process of generating model instances, but is, post-generation, changed in one or more models to variations such as *RTfrankfurtAmMianMunich* (note the spelling mistake: "Mian" instead of "Main"). Synonyms are extracted from the synsets provided by the WordNet® database. Similar variations are done for all generated names.

In addition to varying names, incompletenesses are also introduced: at random, units associated with length properties of segments that are semantically equivalent are removed. This is not considered an introduction of an inconsistency, since it represents a mere omission of information. In addition to removing units, the length and unit of length are also converted between different units and unit systems at random. This is done prior to removing units to allow for cases where two semantically equivalent properties have a different numeric value for the length due to a prior conversion between units (say from meters to feet for one of the units). Such a case would be present if, for instance, in figure 41 the property *lengthUnit* were omitted from the segment *StuttgartToUlm*. Such random introductions of incompletenesses and errors are introduced to convolute the problem of identifying semantic equivalence.

Finally, inconsistencies are injected at random into the model. Primarily, inconsistent constraints on properties are introduced. For the length property, this is done by adding a random number to the length, or by randomly changing the unit (or both). For switch states, a random switch state is chosen for one of the equivalent switch state properties (one of which is determined randomly). Due to the associated challenges, inconsistencies in semantically equivalent properties are the primary type of inconsistency considered. Identifying the semantic equivalence and possible state of inconsistency of two *length* properties requires analyzing the semantic context, such as the possibility that the owning *segments* are semantically equivalent – a challenge, given the potentially very large number of segments in one set of models. Similar to the introduced semantic equivalences, all injected inconsistencies are stored in a list, which is *not* made available to the Bayesian reasoning algorithm.

8.2.2.5 Translation to, and Representation in RDF

Once generated, the models and meta-models are transformed to a graph formalism by translating the Java objects and classes to an appropriate RDF representation. As mentioned in section 8.1, RDF is an appropriate set of specifications that matches the desired characteristics of the common graph-based representational formalism for heterogeneous models developed in chapter 5. The translation is done algorithmically through a set of reflective rules. The reflective RDF counterparts of the respective Java constructs utilized are summarized in table 7. Note that *rdf*, *rdfs* and *xsd* refer to the namespaces for RDF, RDF Schema and XML Schema Definition [227], respectively.

Using the rules from table 7, every Java class representing a meta-level construct is converted to a corresponding RDF/RDFS resource of type *RDFS Class*. The general pattern followed is that every meta-class is translated to a resource of type *RDFS Class*, and every related relationship (e.g., attributes, links and relationships) is translated to a resource of type *RDF Property*. Inheritance relationships, such as those between *Segment* and *Switch*, and *TrackElement*, are represented by utilizing

Table 7: Reflective translation rules of meta-model and instance elements from Java classes and objects to RDF constructs.

Java Concept	**RDF / RDFS Concept**
Class	rdfs:Class
Class Inheritance	rdfs:subClassOf
Attribute / Field	rdf:Property
Owning Class of Attribute	rdfs:domain
Attribute Value Type	rdfs:range
Object	rdfs:Class
Primitive Data Type Instances	rdfs:Literal
Primitive Data Type	xsd:string, xsd:int, ...
Class-Object Instance Relationship	rdf:type

the *RDFS subClassOf* property. The domain and range of attributes (and generally relationships) are expressed using *RDFS Domain* and *RDFS Range* predicates. Enumeration classes, such as `LengthUnitKind`, are also translated to a resource of type *RDFS Class*. Enumeration literals are represented as resources with a *value* and an *ordinal*, where the *value* is the enumeration literal string (e.g., `METERS`), and the ordinal a numeric value. To identify the constructs of the railway meta-models uniquely, the namespace `http://railway/ns#` is used. To this namespace, the name of the meta-class and, in the case of attributes and relationships, both the name of the owning class and the name of property are appended to generate a unique URI (e.g., the attribute *length* of class `Segment` is assigned the URI `http://railway/ns#Segment/length`). Note that, for simplicity, the same namespace is used for all meta-models. The result of translating the railway track network meta-model to RDF is shown in figure 42.

Elements of the generated models (i.e., instances of the respective meta-model elements) are also given unique URIs as identifiers. These URIs are generated based on the name of the associated model and model domain (see section 8.2.2.3), as well as

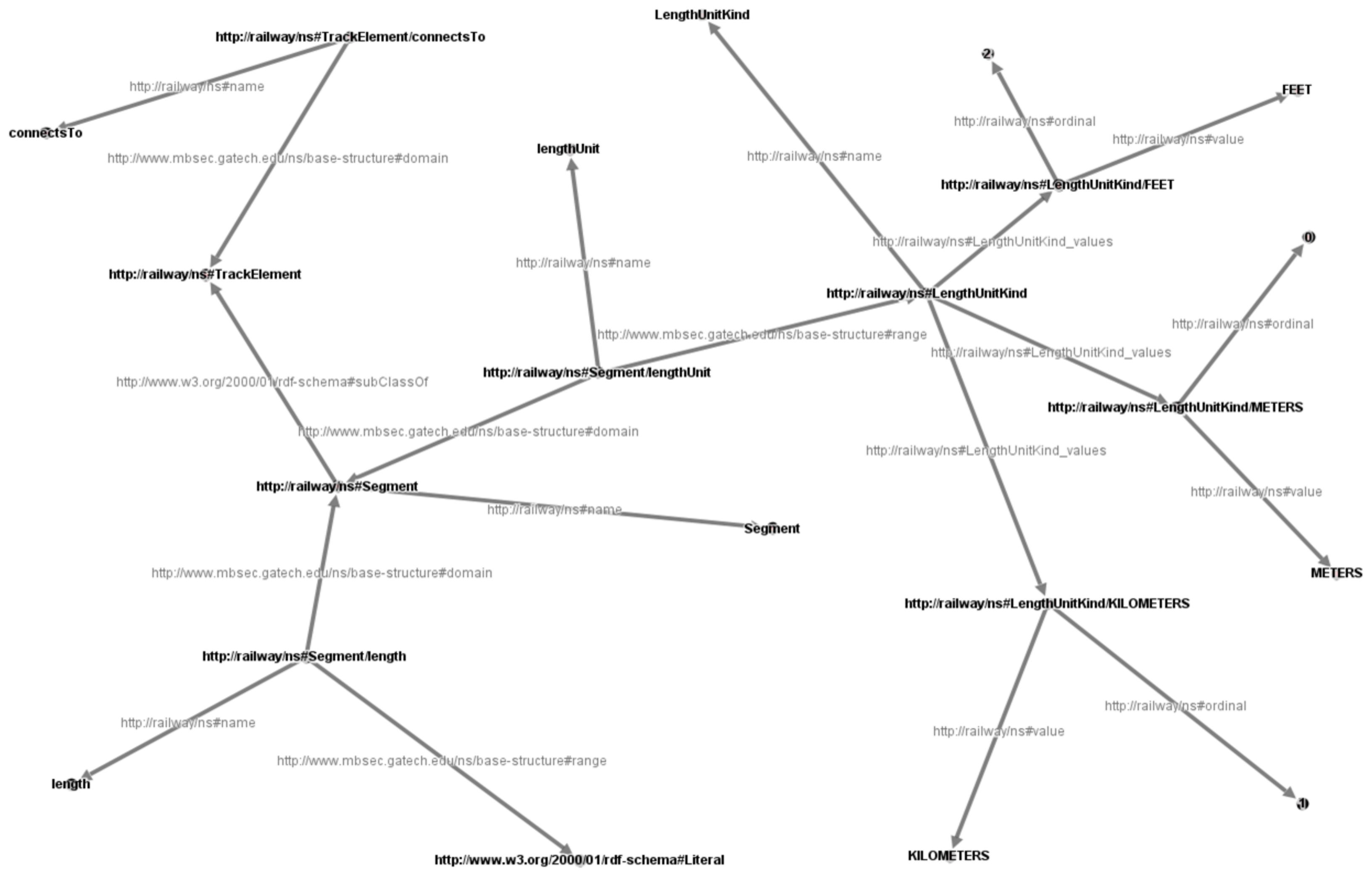

LengthUnitKind
http://railway/ns#TrackElement/connectsTo
http://railway/ns#name
connectsTo
http://www.mbsec.gatech.edu/ns/base-structure#domain
lengthUnit
http://railway/ns#name
2
http://railway/ns#ordinal
FEET
http://railway/ns#value
http://railway/ns#LengthUnitKind/FEET
http://railway/ns#LengthUnitKind_values
http://railway/ns#TrackElement
http://railway/ns#name
http://railway/ns#LengthUnitKind
0
http://www.mbsec.gatech.edu/ns/base-structure#range
http://railway/ns#ordinal
http://railway/ns#LengthUnitKind_values
http://railway/ns#Segment/lengthUnit
http://www.w3.org/2000/01/rdf-schema#subClassOf
http://railway/ns#LengthUnitKind/METERS
http://www.mbsec.gatech.edu/ns/base-structure#domain
http://railway/ns#LengthUnitKind_values
http://railway/ns#Segment
http://railway/ns#value
http://railway/ns#name
Segment
http://www.mbsec.gatech.edu/ns/base-structure#domain
METERS
http://railway/ns#LengthUnitKind/KILOMETERS
http://railway/ns#Segment/length
http://railway/ns#ordinal
http://railway/ns#name
http://railway/ns#value
http://www.mbsec.gatech.edu/ns/base-structure#range
length
1
KILOMETERS
http://www.w3.org/2000/01/rdf-schema#Literal

Figure 42: Graph depicting the RDF representation of the meta-model for railway track networks introduced in section 8.2.2.2 (generated using the visualization tool *Gephi* [13]).

the generated unique ID. Instances of relevant attributes (and relationships) outlined in the meta-models (see figures 36 to 38) are created for each individual instance of a meta-class. This is equivalent to the creation of a slot for a particular attribute value. The generation of the RDF data follows a similar pattern to that of the meta-model, with the exception that the created resources are instances of the respective meta-model element resources rather than just the relevant RDF/RDFS element. An example translated instance of a `Segment` and its *length* and *lengthUnit* attributes is illustrated in figure 43.

Note that the name of each element translated to RDF is made explicit using a special predicate *name* in the railway namespace (`http://railway/ns#name`). This is done for both meta-model elements and their instances, as well as any attributes, relationships and slots.

The generated RDF representations of the models are typically highly complex and can be non-intuitive to a human if not abstracted by filtering out details. Figure 44 illustrates a generated track network model, showing only resources that are segments and switches, as well as their respective connections. Meta-model information, as well as the other attributes (length, length unit and switch state), have been left out. Such figures have been created for the purpose of verifying both that the generative algorithm produces instances that conform to the desired properties (e.g., branching off at switches, ring of routes (see section 8.2.2.3)), and that the defined mapping to RDF works as intended.

8.2.2.6 Mediation to Base Vocabulary & Unit Conversion

The following inference rules are used for mediating both the meta-model elements and instances thereof to the base vocabulary: all meta-classes (and instances thereof) except enumeration classes are assigned the additional type *base:Element*. All attributes (and relationships) and slots are assigned the additional type *base:Property*.

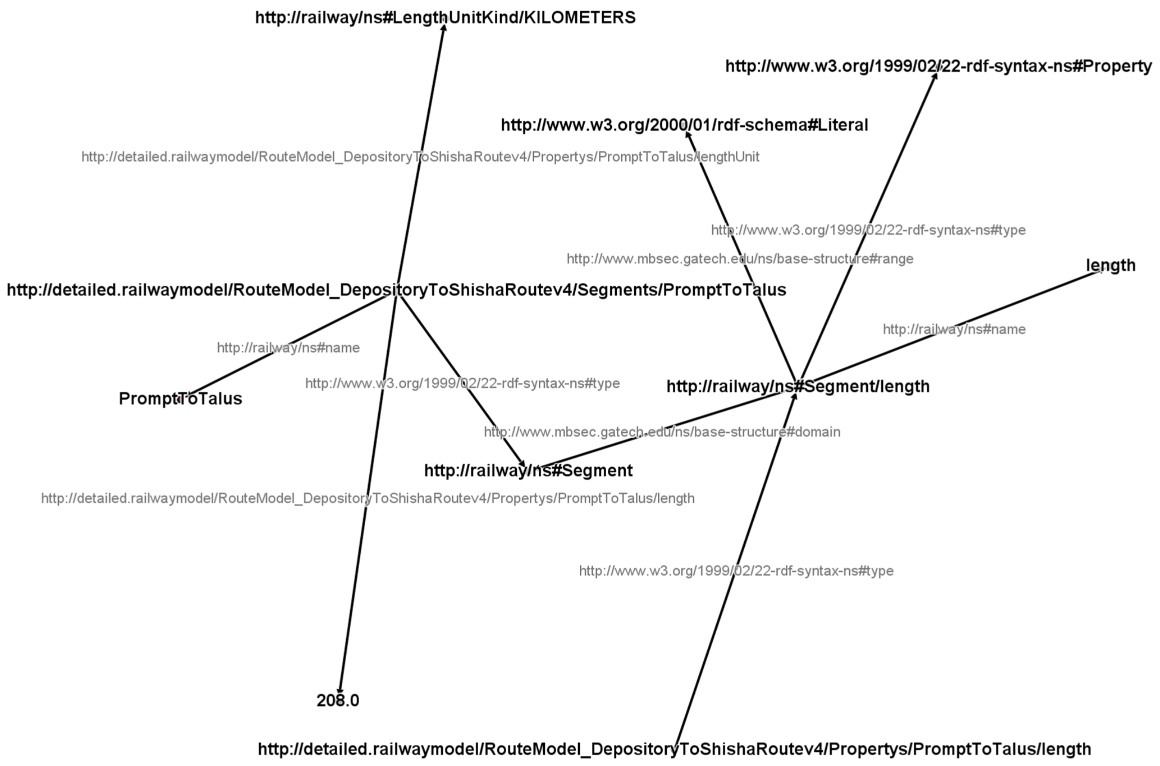

http://railway/ns#LengthUnitKind/KILOMETERS
http://www.w3.org/1999/02/22-rdf-syntax-ns#Property
http://www.w3.org/2000/01/rdf-schema#Literal
http://detailed.railwaymodel/RouteModel_DepositoryToShishaRoutev4/Propertys/PromptToTalus/lengthUnit
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.mbsec.gatech.edu/ns/base-structure#range
length
http://detailed.railwaymodel/RouteModel_DepositoryToShishaRoutev4/Segments/PromptToTalus
http://railway/ns#name
http://railway/ns#name
PromptToTalus
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://railway/ns#Segment/length
http://www.mbsec.gatech.edu/ns/base-structure#domain
http://railway/ns#Segment
http://detailed.railwaymodel/RouteModel_DepositoryToShishaRoutev4/Propertys/PromptToTalus/length
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
208.0
http://detailed.railwaymodel/RouteModel_DepositoryToShishaRoutev4/Propertys/PromptToTalus/length

Figure 43: Graph depicting the RDF representation of an instance of a `Segment` and related slots for `length` and `lengthUnit` (model unique identifiers are not shown in URIs to conserve space; visualized using *Gephi* [13]).

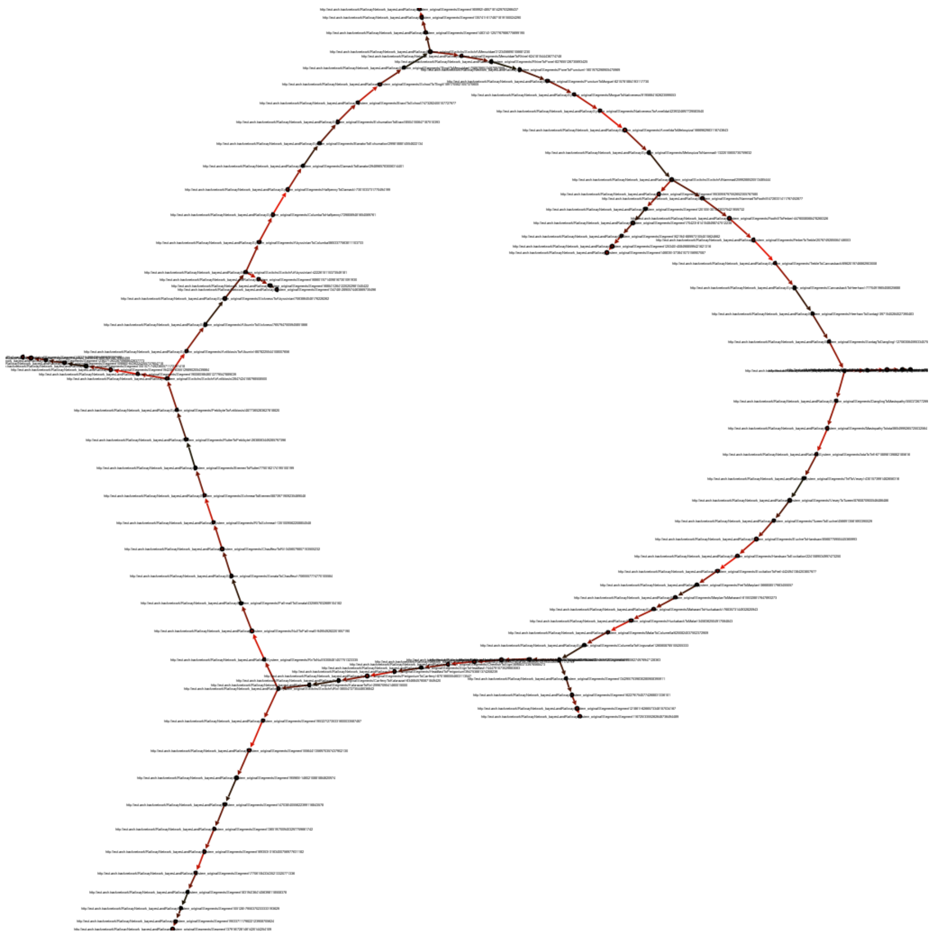


Figure 44: Example of a generated track network transformed to a RDF graph (only segments, switches and connections are shown; visualized using *Gephi* [13]).

Names expressed using the *name* predicate from the railway namespace are mediated to *base:name* predicates. Similar to the case presented in detail in section 5.5.2, this means that, by inference, all elements carrying a statement with the predicate *name* from the railway namespace have a semantically equivalent statement with predicate *base:name* which carries the same literal value as the object.

Similar to the description of the mediation to the base vocabulary given in section 5.5.3, slot values are mediated to constraints imposed on a property. The process is identical to that illustrated in figure 24. The length of a segment is treated as a special case, since a related property defining the unit may exist. Therefore, the length and length unit are aggregated to a single length property with the corresponding constraint carrying a value for the *base:unitType* that is identical to the value assigned to the *lengthUnit* slot value.

Equivalent base constraints (see section 5.5.5) are inferable for constraints for which sufficient knowledge for a unit type conversion exists (in the form of a logical implication / rule). Results from the *Quantities, Units, Dimensions and Data Types Ontologies* (QUDT) project[3] (developed by TopQuadrant and the NASA AMES research center) were utilized as a basis for defining types of units (e.g., feet, kilometers and inches) and conversion multipliers to a base unit (e.g., meters). QUDT is a collection of ontologies that define base classes, properties, and instances for modeling physical quantities, units of measure, and their dimensions in various measurement systems. QUDT uses the expressiveness of the *Web Ontology Language* (OWL) to provide for automated conversion (given an appropriate interpretor). These statements were removed, and only the core statements, which are written in a format that conforms to RDF, were used.

For the purpose of mediating the translated RDF data, a number of inference rules have been written. The syntax of these inference rules corresponds to that accepted

[3]http://www.linkedmodel.org/catalog/qudt/1.1/index.html

by the *Generic Rule Reasoner* from the Jena framework (which is an implementation of Datalog). The exact rules are not presented in this document due to their sheer length and complexity. However, a copy may be requested from the author of this dissertation.

8.2.3 Infrastructure & Environment Setup

For the purposes of generating (inconsistent) models and collecting data, a number of Java-based programs and classes were implemented in addition to the reasoning infrastructure introduced in section 8.1. These include a parameterized generator of railway models, a program that transforms railway models into corresponding RDF representations, data storage capabilities (for storing gathered data points in Microsoft Excel (.xls) format) and post-processing functions. In addition, a number of functions have been implemented in Matlab to post-process and visualize results. To ensure reproducibility of the results, the software versions used, as well as the specifications of the hardware utilized for running the experiments is documented in the following. In addition, relevant enhancements to the reasoning infrastructure are summarized.

8.2.3.1 Software Versions Used & Hardware Configuration

All proof-of-concept software was implemented and tested in the development environment Eclipse Indigo (build 20110615-0604). The compiler, language specification and libraries associated with Java Development Kit (JDK) 7 (1.7.0_01, 64 bit) were used in writing all software. Maven 3.2.3 was used for managing Java project dependencies. Windows 7 (64-bit) was used as an operating system environment. In addition to the reasoning infrastructure and `JBayNeT`, the following external libraries (with their respective versions) were used:

- JUnit 4.11

- Apache Jena 2.10.1
- Apache Fuseki 1.1.1
- Apache Commons Lang 3.3.2
- Apache Commons Math 3.2
- Apache Commons Collections 3.2
- Apache POI 3.11
- Apache Log4J 2.0-rc1
- JGraphT 0.9.1
- MapDB 1.0.6
- JavaBayes 0.346
- WordNet 3.1 (database files only)
- RiTa 1.0.68
- JAWS 1.3

All experiments were performed on a standard office PC with an Intel®Core^{TM}i7-2600 CPU running at 3.40GHz with 16GB DDR2 RAM and a 7200 rpm hard disk.

8.2.3.2 Implemented Software Performance Enhancements & Critical Resource Utilization Mitigation Strategies

Because of the expected long runtime of the reasoning algorithm (see section 6.3.1) due to the expected complexity of the operations performed (primarily pattern matching and Bayesian network inference), `JBayNeT` was configured to use the junction tree algorithm by default for Bayesian network inferences. In addition, to save computation cycles, the reasoning engine was provided with *hints*. These hints take the form of a heuristic, which results in the reasoning engine skipping an attempt to match patterns associated with random variables that are deemed *unobservable* (e.g., the patterns associated with the random variables *InconsistentProperties* and *EquivalentProperties* (see figures 62 and 46)).

In addition to reducing CPU cycles, an effort has been made to reduce the memory consumed during each execution of the reasoning engine. Mainly, this was done through an additional hint, which forces the reasoning engine to store inferences related to only a select number of random variables (e.g., inferences about inconsistency and semantic equivalence). Because the information stored for this purpose is expected to grow with $O(n^2)$, the hash tables used for storing the information are partially cached on the hard disk using cached tree maps from the project *MapDB*. While the latter will result in reduced computational performance, it enables the processing of larger models.

8.2.4 Evaluation Metrics

One of the primary goals of the quantitative evaluation is to collect sufficient evidence to make assertions about the performance of the Bayesian approach to inconsistency identification introduced in chapters 6 and 7. Collecting such evidence and making assertions about qualities of the approach requires the collection of data points and their interpretation which, in turn, requires performance metrics. A basis for the accompanying measurements is provided by the already-introduced generation of overlapping heterogeneous models of railway systems which are injected with inconsistencies (see section 8.2.2). In the following, relevant metrics are introduced that provide a basis for making assertions about various qualities of the approach and define the data to be collected.

Table 8: Table summarizing the definitions of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). This is also known as the *confusion matrix*.

		Actual Condition	
		Positive	*Negative*
Test / Classification	*Positive*	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

8.2.4.1 True Positives, False Positives, True Negatives and False Negatives

A basis for (most) measures of quality of a classifier for a two-class prediction problem is the number of correctly and wrongly classified elements. For the Bayesian approach to inconsistency identification, a *correct classification* means one of two things: either an outcome (e.g., a pair of properties) was correctly classified as being *inconsistent*, or correctly classified as being *not inconsistent*. A *wrong classification* refers to one of the following: either an outcome was classified as inconsistent, even though it is not, or an outcome was classified as not inconsistent even though it is. Here, the term *correctly* and *incorrectly* refer to the *actual state*, and *classified* refers to the outcome of applying a decision heuristic (e.g., *if the probability of inconsistency of a particular pair of properties given the observed evidence is greater than a cutoff value c, then the pair of properties is classified as inconsistent*).

Elements that are correctly classified are referred to as *true positives* (TPs) and *true negatives* (TNs). Within the context of the Bayesian approach to inconsistency management, a TP corresponds to a correctly identified inconsistency. A TN corresponds to the case where an outcome was correctly identified as being not inconsistent. The third and fourth cases are known as *false positives* (FPs) and *false negatives* (FNs). The result of a classification produces a FP if it identifies an outcome as inconsistent, even though it is not. A FN is the result of classifying an outcome as not inconsistent even though it is in fact inconsistent. As discussed in section 7.3 the last case has the most impact on utility. In statistics, the production of a FP leads to a *type I error*. *Type II errors* are defined by the occurrence of a FN. From the Bayesian point of view, a type I error is one that looks at information that should not substantially change a prior belief, but does. A type II error is one that looks at information which should change a prior belief, but does not.

The notion of TP, FP, TN and FN is summarized in table 8. This table is also known as the *confusion matrix* [124]. There, the term *"positive"* may be replaced with *"inconsistent"* or *"semantically equivalent"*, and the term *"negative"* with *"not inconsistent"* or *"semantically different"*[4]. Note that, within the context of inconsistency identification, the sum of the number of true positives and false negatives must, by definition, equal the number of actual inconsistent (or semantically overlapping) outcomes. Similarly, the sum of the number of false positives and true negatives must correspond to the number of outcomes that are not inconsistent (or not semantically overlapping).

8.2.4.2 Derived Metrics

Various metrics can be derived from the notions of TPs, TNs, FPs and FNs. Relevant metrics from the related literature include the *recall*, *precision*, *specificity*, *F-measure* and *fallout*. These metrics are introduced in the following pages.

[4]The terms *"positive"* and *"negative"* are commonly used within this context due to their roots in medical research, and their relation to diagnosing diseases by using the outcome of tests as evidence. There, as already introduced in chapter 6, the use of Bayesian reasoning is commonplace.

Recall Recall is also known as the *sensitivity* in biomedical research, and sometimes referred to as the *hit rate* or *true positive rate* in machine learning applications. It measures the number of TPs in relation to the number of *actual* cases [124]. In other words, it measures the *accuracy* of a classification heuristic, since it measures the number of correctly identified elements in relation to the actual occurrences. Within the context of inconsistency identification this means that a recall gives an indication of what fraction of actual inconsistencies were identified. Therefore, recall can be defined in the following way:

$$\text{recall} = \frac{TP}{TP + FN} \tag{21}$$

A recall value of 1.0 is indicative of a classification heuristic with perfect accuracy. In other words, a recall of 1.0 indicates that no false negatives are produced. A recall of 0.0, on the other hand, indicates that no correct classifications were made. Note that a recall value of 1.0 is easily achievable by setting the cutoff value to 0 (i.e., if every outcome is classified as inconsistent). However, as discussed in section 7.3 this is not very useful, since a large number of false positives would be produced, which potentially incurs a very high cost. Therefore, additional measures must be considered when evaluating the performance of a classifier.

Precision Related to recall is the notion of *precision.* Precision is a measure of how well a classifier is able to distinguish between TPs and FPs [124]. Within the context of inconsistency identification, precision is a measure of the fraction of outcomes classified as *inconsistent* which in reality were inconsistent. Precision is also known as the *positive predictive value* and is defined in the following way:

$$\text{precision} = \frac{TP}{TP + FP} \tag{22}$$

Similar to recall, a high precision value is desirable, since a high value is indicative of a low number of false positives relative to the number of true positives. However, precision does not take into account the number of false negatives which, as discussed in section 7.3 typically incur a much higher cost than false positives. Therefore, high values in precision can be achieved with high cutoff values, since the number of false positives expectedly declines with the cutoff value.

Specificity *Specificity* is a measure of performance of a classifier that considers the proportion of negatives which are correctly identified as such [66]. Within the context of this research, specificity refers to the fraction of outcomes that were correctly classified as being inconsistent (or semantically overlapping). Therefore, specificity can be defined as the number of true negatives, divided by the sum of the number of false positives and true negatives. Specificity is the true negative rate.

$$\text{specificity} = \frac{TN}{FP + TN} \tag{23}$$

Within the context of this research, a classifier with specificity 1.0 identifies all outcomes that are not inconsistent (or not semantically overlapping) as such. The value is high if the classifier can correctly classify non-inconsistent outcomes well.

Fallout Complementary to specificity is the fallout rate (sometimes also referred to as the *false positive rate*) [66, 124]. It measures the proportion of positives correctly identified as such. Therefore, within the context of inconsistency identification and semantic overlap detection, the fallout rate is the fraction of outcomes that were wrongly classified as being inconsistent, divided by the total number of non-inconsistent outcomes.

$$\text{fallout} = \frac{FP}{FP + TN} = 1 - \text{specificity} \tag{24}$$

A good classifier should have a low fall-out rate, since it is indicative of a low number of cases in which an outcome was wrongly classified as being inconsistent. However, as with other measures, the fallout rate must be considered in relation to other measures and the cost associated with incorrect classifications.

F-Measure One commonly employed measure in the related literature on information theory, ontology matching, and database schema matching is the *F-measure* (or *F1-score*, *F-score*). It is a derivative of the effectiveness measure proposed by Van Rijsbergen in [223], and its general form is:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{25}$$

Typically, the F-measure is defined as the harmonic mean of precision and recall. The harmonic mean puts the same emphasis on precision and recall, and can be formed by setting $\beta = 1$:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{26}$$

More emphasis is put on precision by setting $\beta < 1$, and more emphasis on accuracy with $\beta > 1$. The F-measure is a standard measure used in identifying an adequate value for the classification cutoff. Its widespread use is due to it taking into account the trade-off between precision and recall. However, since the value for β is an arbitrary weight, it is merely a heuristic. Therefore, and as can be concluded from the discussion in section 7.3, better results can likely be achieved by taking into account the costs associated with producing FPs and FNs, and the costs of polling external information sources.

8.2.4.3 Receiver Operating Characteristic

To judge the performance of a classifier, the *receiver operating characteristic* (ROC) (also known as the ROC curve) is typically employed in machine learning applications [66]. The ROC curve is simply a plot of the intuitive trade-off between sensitivity and specificity, with the horizontal axis flipped for historical reasons. Therefore, the ROC is a plot of the sensitivity against the fallout ratio (1 - specificity) for various cutoff values. ROC analysis provides the most comprehensive description of diagnostic accuracy because it estimates and reports all of the combinations of sensitivity and specificity that a classification is able to provide [61].

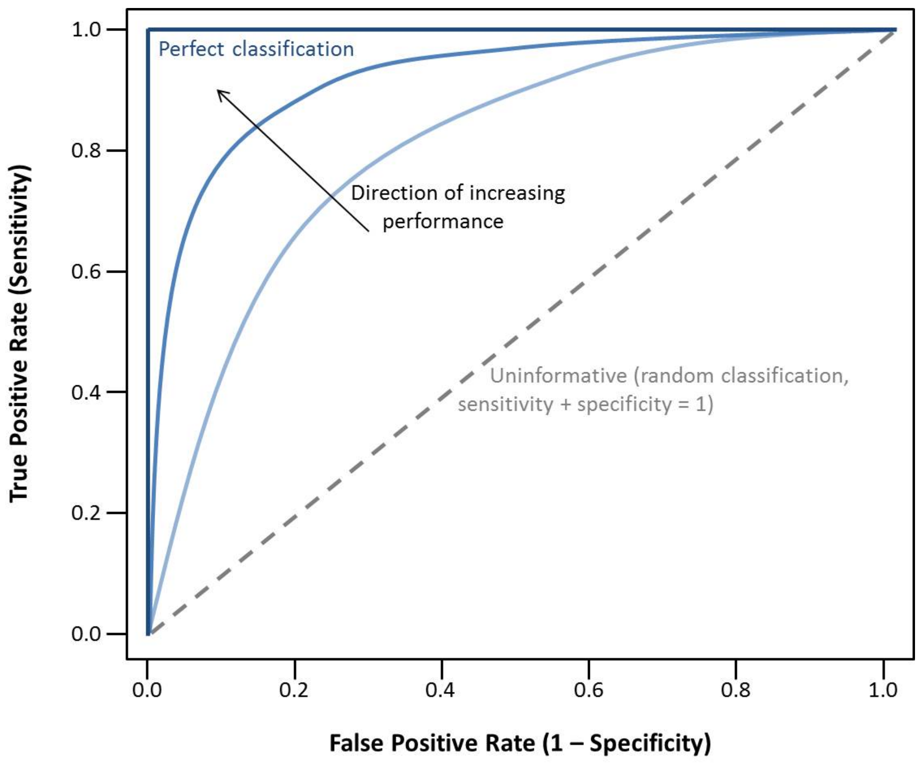


Figure 45: Qualitative representation of several receiver operating characteristic (ROC) curves. The ROC is commonly used in evaluating machine learning classification performance.

A qualitative ROC curve is illustrated in figure 45. If sensitivity and specificity is equal for all cutoff values, the classification is considered uninformative and random. If the observer was completely uninformed, then the ROC curve would be a straight line connecting the lower left to upper right corners (see figure 45, and the area under this curve would be 0.5. This line then corresponds to a 50% probability of the observer correctly classifying a random outcome. In the related literature, the area under the curve is considered a measure of *accuracy* (sometimes referred to as the *average accuracy*) or *goodness* of a classifier: it is indicative of the rate of distinguishing a true positive from a false positive. Therefore, the larger the area, the "better" a classifier. Perfect classification is indicated by an area of 1.0. The area under the curve is interesting from the perspective that it is a measure for the accuracy of the classifier that is independent of the cutoff value chosen when interpreting the inference result.

ROC analysis addresses the variance of sensitivity and specificity due to variance in interpretation thresholds. However, it is still subject to some limitations: first, only binary classifications can be considered, such as the presence or absence of inconsistencies. Secondly, ROC analysis still requires a reference standard that indicates the true state (e.g., whether a pair of properties is inconsistent or not). Imperfect references standards would introduce a bias. In addition, the area under a ROC curve is not a good measure of performance if the cost of misclassifying examples in one class (e.g., *"is inconsistent"*) is very different from misclassifying examples in the other class (e.g., *"is not inconsistent"*), or the occurrence of one class is much rarer than the other [47].

8.2.4.4 Value-Based Comparison of Classifiers

In this section, an alternative measure of performance for a binary classifier based on its *value* is introduced. The value of a classifier can act as a basis for comparison of two or more alternative classifiers. Determining the value of a classifier requires taking into consideration the value of the *mission* – here: the identification of inconsistencies – and subtracting the invoked cost. This invoked cost depends on the *cutoff value*, the cost of testing, and the number of correctly and wrongly classified elements. One way of determining the value of a classifier is to calculate the expected value of classifying a single outcome at a particular cutoff value $c = i$:

$$\mathbb{E}\left[V_{c=i}\right] = \mathbb{E}\left[V_{mission}\right] \; - \; C_{inv,c=i} \tag{27}$$

The invoked cost $C_{inv,c=i}$ is dependent on the expected cost of performing the test (i.e., collecting the evidence about the outcome from the graph-based model by matching various patterns), the expected cost of verifying a true positive $C_{ver,TP}$, the expected cost of verifying a false positive $C_{ver,FP}$, and the expected cost incurred by a false negative C_{FN}. Determining the expected cost incurred by a classifier also requires knowledge about the *(true) expected frequency* of TPs, FPs and FNs occurring at various cutoff values. These frequencies are denoted by $P_{TP,c=i}$, $P_{FP,c=i}$ and $P_{FN,c=i}$. Note that the cost is not dependent on true negatives. Therefore, the expected cost incurred per sample can be expressed as follows:

$$C_{inv,c=i} = C_{test} + P_{TP,c=i}C_{ver,TP} + P_{FP,c=i}C_{ver,FP} + P_{FN,c=i}C_{FN} \tag{28}$$

Note that one can assume that $C_{ver,TP} = C_{ver,FP}$ since, in both cases, a verification of an inferred statement is required, and it can be argued that the cost of verification does not depend on whether a true positive or false positive is found. Furthermore, by nature of inconsistencies, the cost of *not* identifying an inconsistency is expectedly much larger than the cost of verification of a TP or FP. Therefore, $C_{FN} \gg C_{ver,TP}$. However, $C_{ver,TP}$ is non-negligible if $P_{FN,c=i} \ll (P_{TP,c=i} + P_{FP,c=i})$. This assumption is deemed acceptable for the general case, particularly in cases where a very large number of inferences are expectedly performed (e.g., in the case with identifying inconsistent pairs of properties), and the inconsistency identification knowledge has been carefully crafted.

To compare two or more classifiers, a number of additional simplifying assumptions can be made. Firstly, for a single application context (i.e., the development of a particular system), the value of the mission can be said to be equal across all classifiers. This leads to the conclusion that a comparison of the classifiers can be based on the incurred cost. Secondly, it can be assumed that C_{test} is constant on average for each sample and in relation to the size of the graph-based model being polled. In summary, given the cost of two classifiers with an identical mission (here: identification of inconsistencies) that are applied in the same context (a single system), the classifier that invokes *the least cost* should be preferred, since, by the assumptions made, it is of the highest *value*.

8.2.5 Experiments & Results

In the following pages, the results from performing a series of experiments are outlined. The primary goal of the experiments is to characterize the behavior of the proposed inconsistency identification approach. This is done by using the proposed approach for the purpose of *identifying inconsistent (and, in the process, overlapping) pairs of properties*. The collected results act as evidence for determining the validity of hypotheses 3, 4 and 5, and aid in answering research questions 2 and 3.

The basis for the measurements in each experiment are a number of automatically generated sets of railway models. The data collected for each experiment is averaged over 35 generated sets of sets of models. Each set of heterogeneous railway models describes:

- 2 to 5 routes
- 4 to 12 segments per route
- 1 to 3 switches per route

- 1 to 3 sensors per generated switch
- 6 to 14 additional track elements per switch in the track network

The exact number is determined by sampling uniformly and independently from each range. In the process, $2 + n$ (where $n = \#\text{routes}$) models are generated (1 system model, n detailed route models, 1 track network model). The probability of *potentially* introducing an inconsistency *to a pair of equivalent properties* is set at 0.3 (recall from section 8.2.2.4 that it is possible for an inconsistency to be intended to be introduced, but because of the circumstances encountered, the inconsistency may not be introduced). Note that 0.3 was chosen based on research by the *National Institute of Standards and Technology* (NIST), which showed that under stressful working conditions humans tend to have an error rate of 30 percent.

For each experiment, the following data is stored for purposes of post-processing:

- The Bayesian network structure and parameters used
- For each generated set of models:
    - A list of the actual semantic equivalences (i.e., model overlap)
    - A list of the actual inconsistencies
    - The specific parameters used in generating the model
    - The raw, generated models in RDF

In addition to the metrics above, experiment-specific measurements and any supplementary (generated) data is stored as well.

8.2.5.1 Initial Setup & Bayesian Network Used

To characterize the proposed approach, and to compare it to existing approaches, the Bayesian network illustrated in figure 46 is primarily used for the following experiments. Note that this network ignores much of the larger semantic context around the

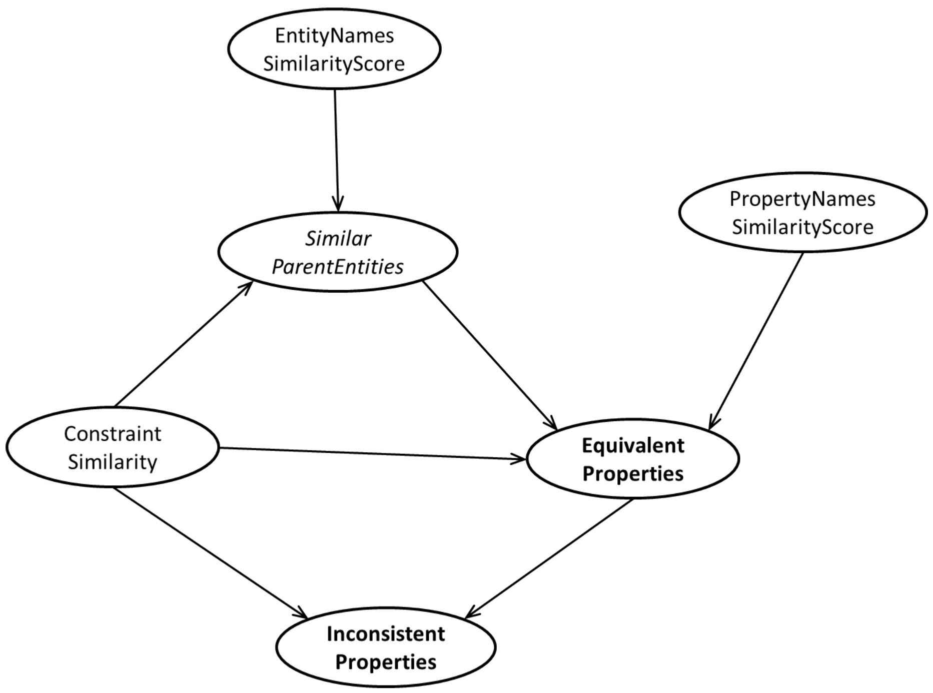


Figure 46: Compact Bayesian network used in reasoning about the inconsistency and semantic equivalence of distinct pairs of properties.

properties, and an analysis of representation conventions and syntactic similarity is used as the primary evidence for semantic equivalence (similarity of the names of the properties, and similarity of the names of the owners of the properties). Whether or not a pair of properties is inconsistent is then determined by the probability of semantic equivalence and by comparing the (base) constraints on the properties. Therefore, the assumptions made are comparable to those made by state-of-the-art automated approaches to identifying model overlap, such as pattern- (or negative constraint-) based inconsistency identification approaches (see chapter 3). The possible target space values and patterns associated with the random variables, and elicited network parameters are detailed in appendix A.1.

The *"similarity score"* referred to in the Bayesian network is a normalized value between 0 and 1 that is calculated based on the *Levenshtein distance* [137] developed (but not exhaustively investigated) as part of this research. The Levenshtein distance measures the difference of two expressions e_1 and e_2 by determining the number of edit operations (insertions, deletions, or substitutions) that are required to transform one expression into another. Therefore, a Levenshtein distance of 0 is indicative of the two expressions being equal, and a distance of n with $|e_1| = n$ and $|e_1| \geq |e_2|$ indicates the highest possible dissimilarity value. The normalized similarity score s of e_1 and e_2 is defined by:

$$s(e_1, e_2) = 1 - \frac{lev(e_1, e_2)}{\max\{|e_1|, |e_2|\}} \tag{29}$$

Note that $\max\{|e_1|, |e_2|\}$ is the *maximum Levenshtein distance* between two expressions. Dividing by this value leads to a normalized form of the Levenshtein distance, where 0.0 indicates equality of the expressions and 1.0 indicates complete dissimilarity. The similarity score introduced here is then simply the inverse of this, where a similarity of 1.0 is defined as complete equality. For the calculation of the Levenshtein distance $lev(e_1, e_2)$, the implementation provided by the *Apache Commons Lang 3.3.2* library (static method in class `StringUtils`) is utilized.

8.2.5.2 Initial Measurements

As an initial starting point, the Bayesian network introduced in the previous section is utilized for identifying inconsistencies (and semantic equivalences) in 35 sets of generated models. *It should be emphasized that no explicit knowledge about semantic equivalences are entered into the models or made available to the reasoner in any way.*

For each generated set of models, all deductions are stored. These deductions are then analyzed to determine the number of true positives, false positives, true negatives and false negatives. This is done by comparing deductions to the list of true inconsistencies and equivalences. By doing so, the number of TP, FP, TN and FN produced *at each cutoff value* can be determined. From these values, the recall, precision, specificity, F-measure and cost incurred are then calculated. Note that only cutoff values between 0.01 and 0.99 (inclusive) are considered, with increments of 0.01.

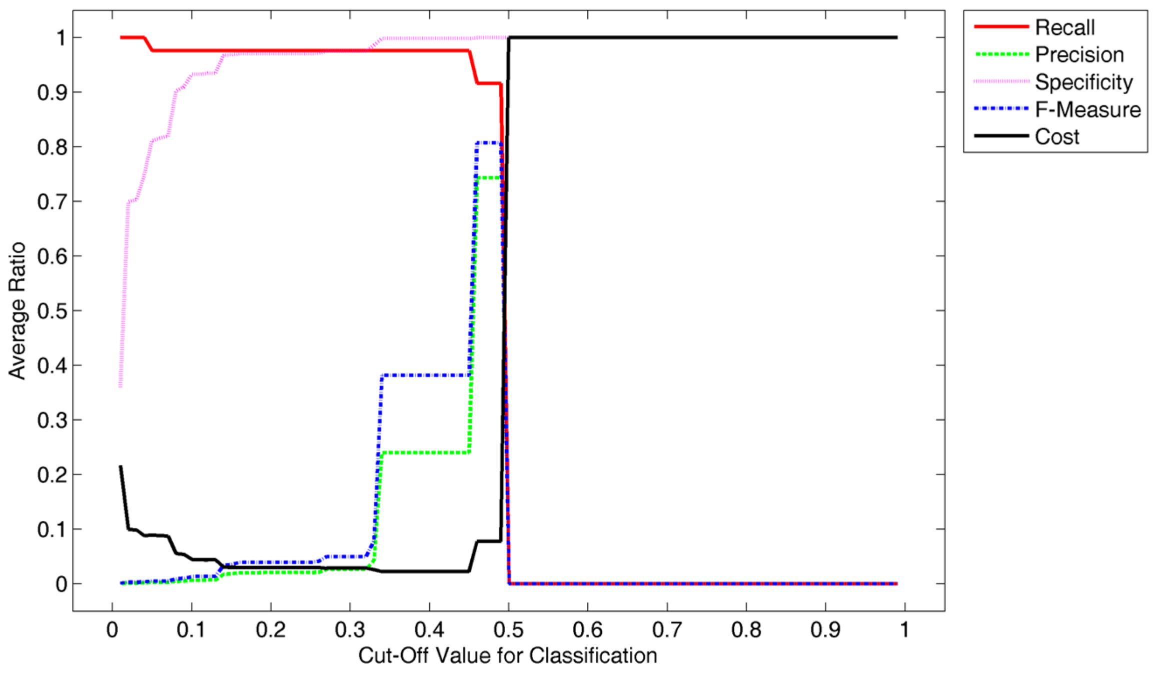


Figure 47: Recall, precision, specificity, F-measure and cost incurred by the classifier for inconsistency identification, averaged over 35 runs and plotted over different cut-off values.

The *averaged* values (over 35 runs) for recall, precision, specificity, F-measure and normalized cost (i.e., cost divided by the maximum cost over all cutoff values) are plotted for classifying *inconsistencies* in figure 47 and for classifying *semantic equivalences* in figure 48. Note that for the cost incurred by an inconsistency, it is assumed that $C_{FN,incon} = 10000\ C_{ver,incon}$ – i.e., the cost incurred by not identifying an inconsistency is 10000 times higher than the cost of verifying a sample manually. For equivalences, it is assumed that $C_{FN,equiv} = 0.05\ C_{FN,incon} = 500\ C_{ver,equiv}$.

Note that in both figures 47 and 48 the incurred cost first decreases, and then increases again. The initial increase in cost can be correlated with the (comparatively) larger number of false positives produced (indicated by the *precision*) if the cutoff value is low. However, the cost rises again once the cutoff value is too large, which is an effect of the classifier producing a larger number of false negatives (indicated by a decreasing *recall*). Note that the F-Measure – a common metric used for determining

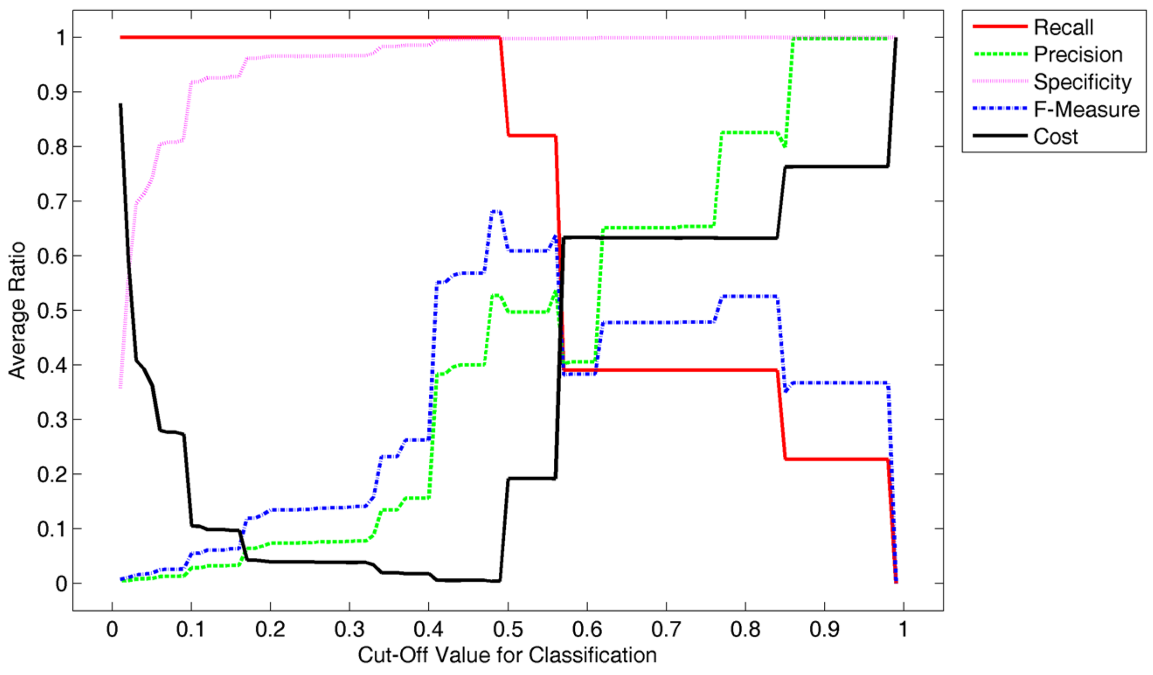


Figure 48: Recall, precision, specificity, F-measure and cost incurred by the classifier for semantic overlap detection, averaged over 35 runs and plotted over different cut-off values for classification.

an appropriate cutoff value – is close to the optimum determined by the incurred cost, but tends to suggest a higher value. However, this can be explained by the fact that equal weight is put on precision and recall in the case of the plotted F-Measure.

Also interesting to note is the very high specificity, even for small cutoff values (note that the specificity is never 0, since 0.0 was not considered as a cutoff value). As mentioned, the number of pairwise comparisons performed by the classifiers is large. Therefore, a high specificity is indicative of a very high number of true negatives as compared to false positives. To illustrate the effect of this on the performance of the classifier, the ROC curve is plotted in figure 49. Note that the large area under the curve (which, visually inspected, is close to 1) is indicative of good performance of the classifier. This is interesting, and somewhat surprising, since relatively naïve assumptions have been made in building the classifier.

Finally, a particularly interesting observation about the inconsistency classifier

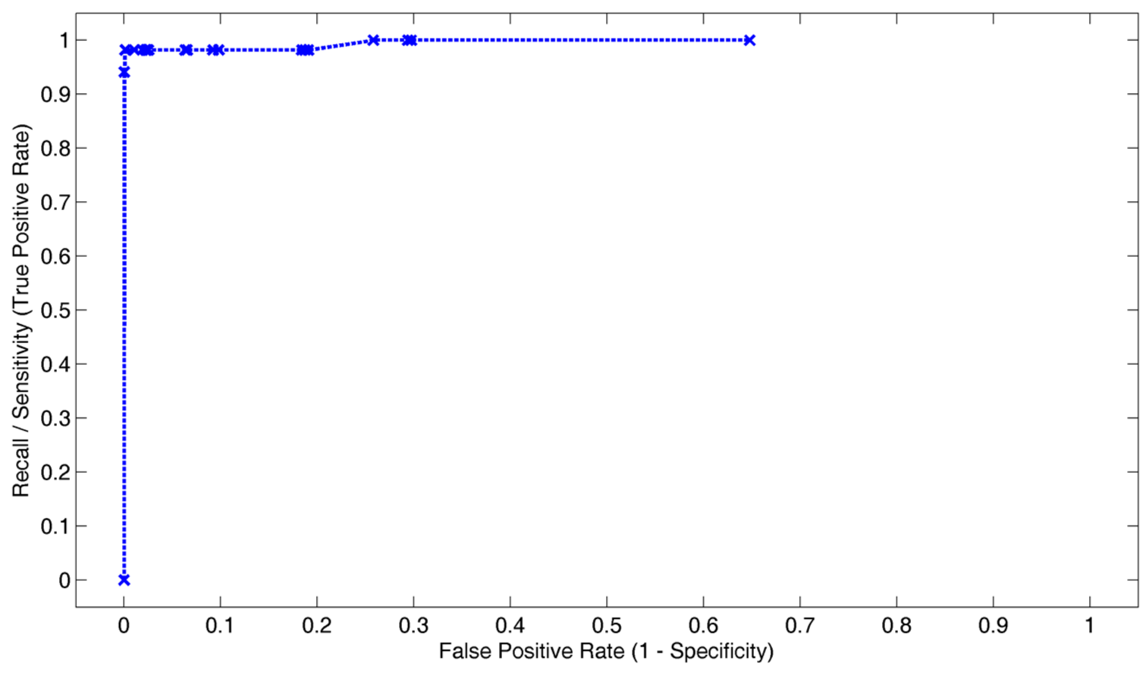


Figure 49: Receiver operating characteristic (ROC) for inconsistency identification averaged over 35 runs (linear interpolation between points).

(figure 47) is that the recall is 0 for any cutoff value ≥ 0.5. However, for the same cutoff value, it is non-zero for the case of classifying equivalences, but there is a significant drop at the same point (figure 48). It seems then that, for cutoff values > 0.49 the evidence for a pair of properties being inconsistent outweighs the evidence for the pair of properties being equivalent. That is, the evidence collected about a pair of properties indicates semantic difference so strongly (due to the degree of inconsistency or, rather, mismatch in information), that the hypothesis of the pair of properties (intendedly) being equivalent and inconsistent is rejected. This supports the discussion from section 7.1.3.

8.2.5.3 Impact of Strong Supporting Evidence for Inconsistency on Semantic Overlap Detection

Based on the observations made in the initial experiment, a hypothesis can be formulated that removing observations about the similarity of the constraints imposed

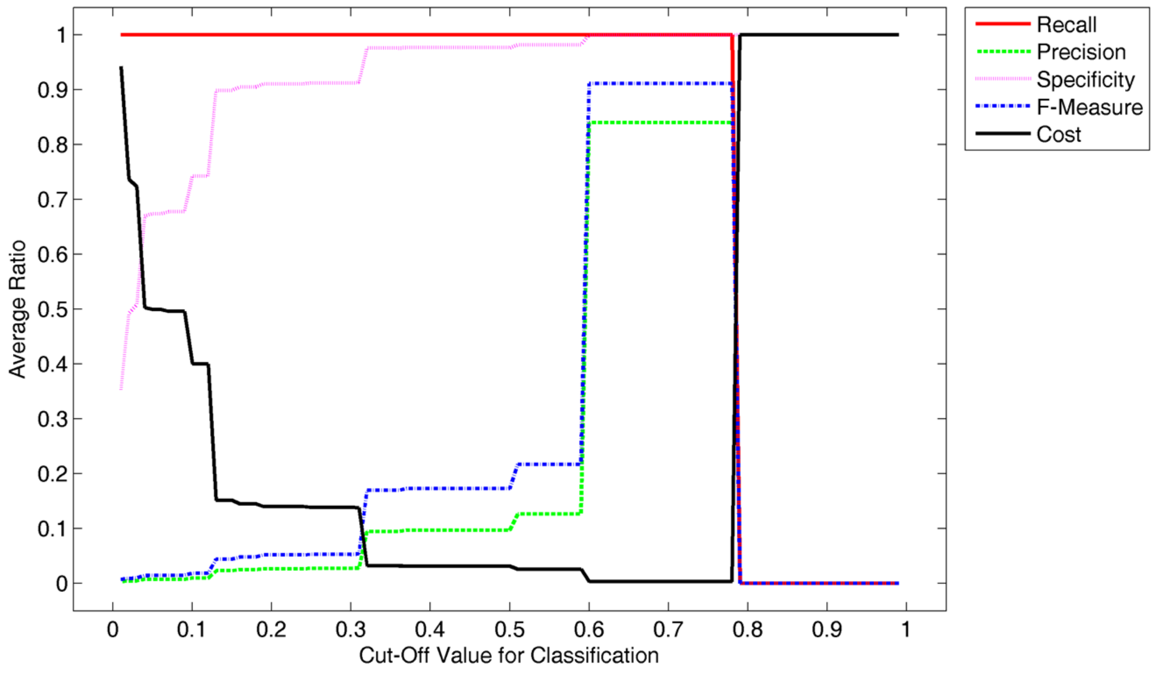


Figure 50: Effect of removing the dependence of constraint similarity on the (intended) semantic equivalence of a pair of properties.

over a pair of properties should lead to a higher recall for a cutoff value of 0.5. To investigate this, the node *ConstraintSimilarity* and the corresponding influence relationship to *EquivalentProperties* is removed from the Bayesian network, the Bayesian network parameters are updated accordingly, and the procedure for measurements as detailed in the initial experiment is repeated. Note that, for reasons of performance, and without impact on the accuracy, the node *InconsistentProperties* was removed also. Figure 50 shows the result for the updated classifier for semantic equivalence.

As suspected, leaving out the comparison of constraints has led to a higher recall value at 0.5. However, it is also interesting to observe that more FPs are produced at lower cutoff values than previously, with a sudden spike of the precision at 0.6, and a sudden downfall of the recall at 0.78. The sudden spike in precision can be explained by the fact that whether or not the owning entities (parents) of the pair of properties are similar provides strong evidence in support of the properties being equivalent, given that the names of the properties are also similar. The sudden decline in the recall can be explained by the fact that, based solely on the name, it cannot be said with certainty whether the owning objects are equivalent (for a score of 0.8 to 1.0 the probability of the parents being equivalent is elicited as 0.8). Combining this with an elicited probability of 0.98 for the properties being equivalent given that their names have a similarity score value of 0.8 to 1.0 and their parents are equivalent, this leads to a value below 0.8. Note that the equivalence of owning parents is never observed explicitly in the graph-based model. Also note that, for brevity, the full set of modified network parameters is not included as a part of this manuscript.

8.2.5.4 Impact and Sensitivity of Prior Beliefs

To this point, a Bayesian network has been utilized for which the network parameters are specified through capturing subjective beliefs on them. However, the resulting probabilities are not likely to be the *true* probabilities (see the discussion in section 7.2.4). Therefore, the impact and sensitivity of the prior beliefs on the network parameters is investigated in this section. The hypothesis is that the average precision and accuracy can be improved by selecting probabilities for the network parameters that are closer to the *true* value.

To investigate the validity of the hypothesis, the true network parameters for the Bayesian network from figure 46 are determined. This is done by comparing the deductions made about all pairs of properties by the reasoner to the true state (as stored during the model generation process). From this, a set of *complete data cases*[5] can be generated. To compute network parameters that are as close to the true ones as possible, 123 sets of models were generated and analyzed. By doing so, $5,516,490$ samples (i.e., unique pairs of properties) were made available. For brevity, the computed network parameters are not included as part of this dissertation document.

[5]In Bayesian learning, a *complete data case* is a data case (see section 2.3.5) in which a value is known for every random variable in the network.

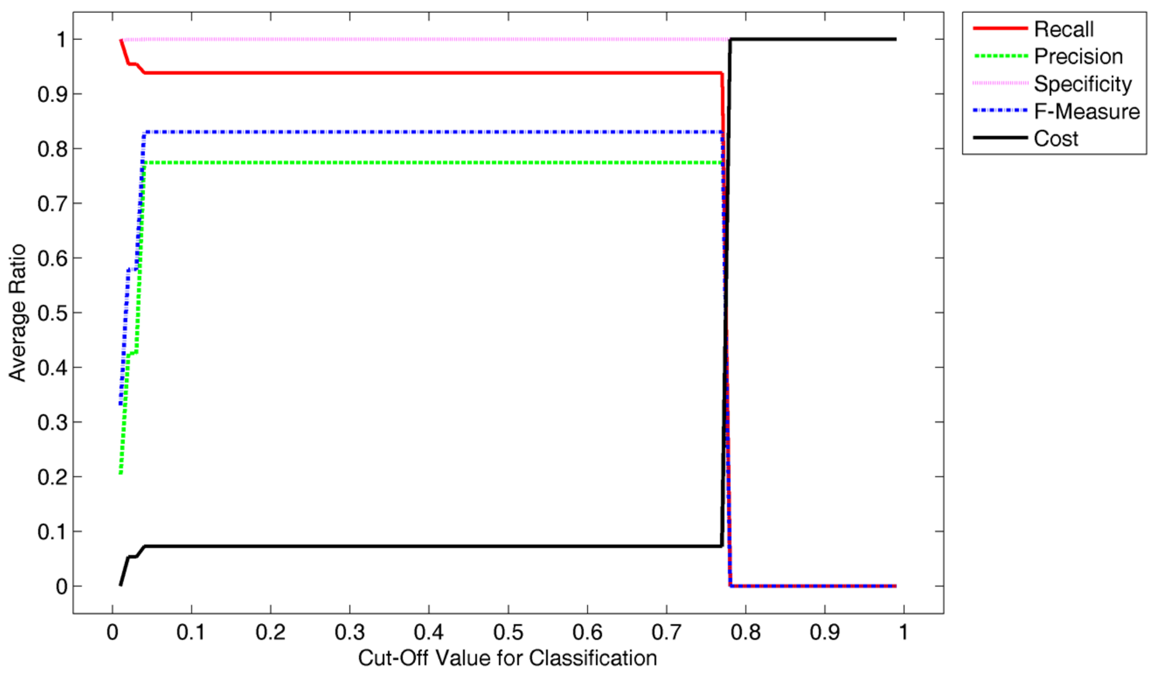


Figure 51: Average recall, precision, specificity, F-measure and incurred cost for the Bayesian network, with *true* values for the network parameters (inconsistency classification).

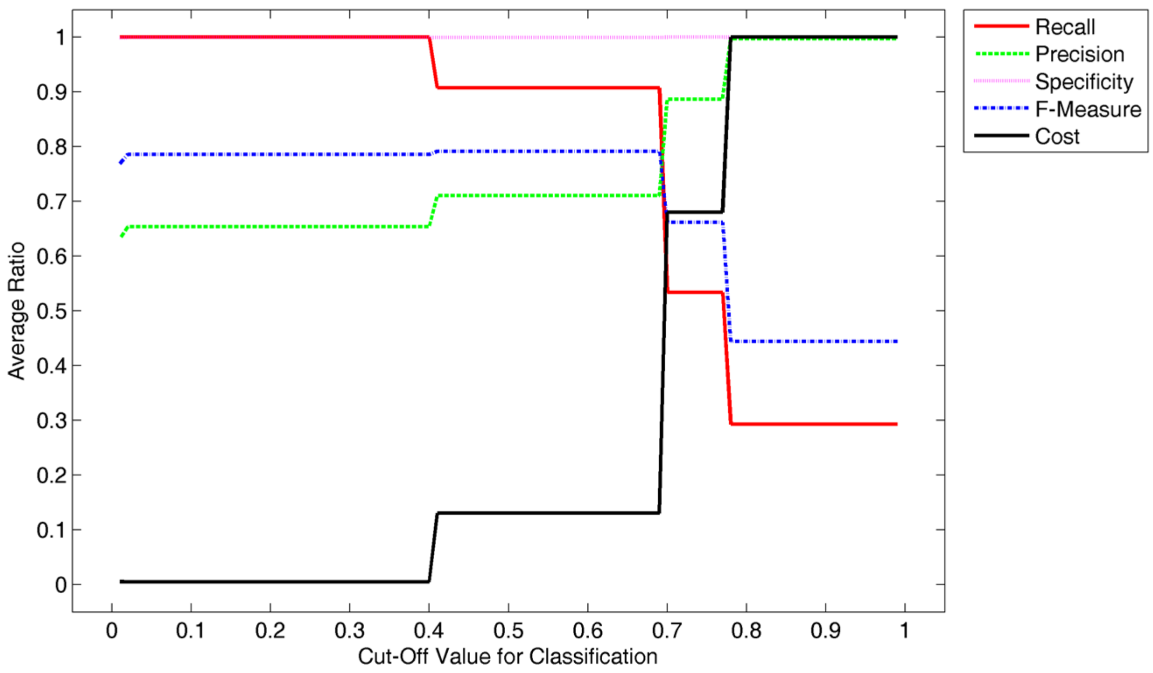


Figure 52: Average recall, precision, specificity, F-measure and incurred cost for the Bayesian network, with *true* values for the network parameters (semantic equivalence).

Using the acquired true network parameters, the recall, precision, specificity, F-Measure and incurred cost are determined using the same procedure as in the initial experiment (averaged over 35 runs). The results are illustrated in figures 51 and 52.

Immediately visible is the very large difference in precision and almost constant, very high value for specificity, over a wide range of cutoff values. Precision drastically rises even with little evidence considered. This strongly indicates that false positives can be reduced by refining the probability distributions on the network parameters. The effect is also visibly dominant in the incurred cost. Here, the curve is monotonic and no longer convex, suggesting a negligible impact of FPs.

Secondly, for the inconsistency classifier, note the almost immediate decline in the recall. The decline happens slightly earlier than the one observed in figure 47. Analyzing the Bayesian network in both cases, it can be observed that the decline seems to coincide with cutoff values above the *prior belief on any pair of properties being inconsistent* (from the Bayesian networks, one can determine this probability (of inconsistency, given no observations) as $p_{initial} = 0.04178$ and $p_{true} = 0.04121$). This may be an indication of an incompleteness (or, rather, underspecification) of the inconsistency identification knowledge (possibly the patterns) which leads to some infrequently occurring manifestations of an inconsistency not being detected. However, this is very likely the result of a relatively rare, but nonetheless possible case when injecting the generated models with inconsistencies. As outlined in section 8.2.2.4, it is possible for the length of two semantically equivalent segments to have the same value, but different *intended* units – that is, no units are specified (in either case), but the modelers intended to use different units (which are not explicitly represented in the model). The hypothesis that this observation is a result of this rare type of inconsistency is supported by the fact that, in figure 52, the recall value for semantic equivalences is still 1.0 at that point. Note that such inconsistencies are almost impossible to identify without further information, and only by choosing a very low

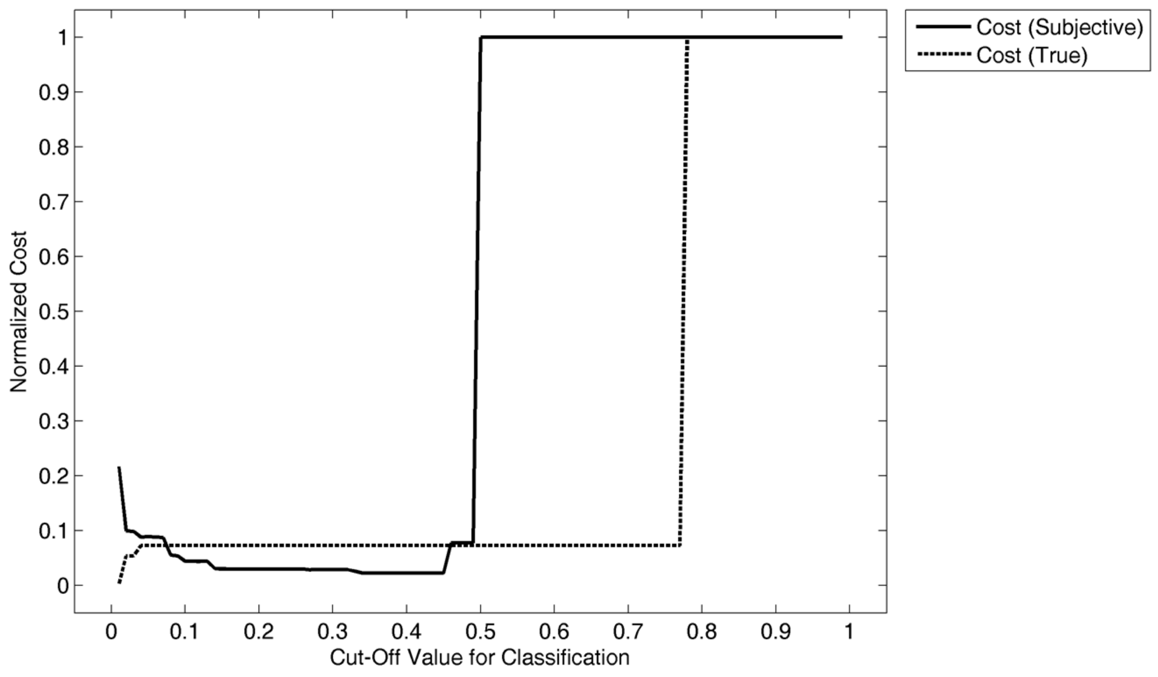


Figure 53: Incurred cost by classification of inconsistencies by the Bayesian network with elicited beliefs on the network parameters compared to the cost incurred using the true frequencies.

cutoff value (here, a cutoff value corresponding to the prior belief) and accepting the production of a large number of FPs.

In the case of the equivalence classifier (figure 52), what is particularly noticeable is that for none of the cutoff values between 0.01 and 0.99 is the recall 0. That is, certain features are considered very strong indications of semantic equivalence. This is an effect of truly *fitting* the parameters to the generated set of models.

It is interesting to compare the difference in value of the classifiers for the cases of using the *true* probabilities and degrees of belief on the network parameters. As mentioned in section 8.2.4.4, the cost incurred by two classifiers can be used as a direct comparison (if the context in which they are applied is the same). Figure 57 is a plot of the cost incurred by the classifier for inconsistencies with elicited subjective beliefs, and the cost incurred by a classifier with the true network parameters over the full range of cutoff values. The costs can be compared since the models are generated under the exact same conditions.

Note that even though a lower minimum cost can be achieved by using a classifier with the true values for the distributions on the network parameters, the classifier with the elicited distributions seems to be more *robust*, and can achieve similar performance for a wider range of cutoff values. It is also capable of correctly classifying inconsistencies in this range at the expense of producing more false positives. This very likely results from a human's tendency to *underweight outcomes*, as is described in section 7.2.4.

Lastly, it should be noted that, given the observations made, the determined parameters are likely not the *true* values, but merely *close to the truth*. An investigation of the network parameters revealed that few data points were available about certain events, suggesting that some events are *very* infrequent, hence producing sub-optimal results. However, the determined parameters are deemed accurate enough to depict the trends, and are, nonetheless, useful in comparing with the case of using subjective beliefs. It is hypothesized that as the number of samples reaches infinity, the values for precision, recall and cost will be constant. A value for the recall or precision below 1.0 would then clearly indicate an under- or over-specification of the *structure* of the network and / or patterns. If recall and precision reach 1, the network and patterns are capable of *entailing* inconsistencies in a logically correct fashion (given the conditions under which the models are generated).

8.2.5.5 Impact of Incremental Learning

From the observations made in the previous experiment, it can be deduced that the production of FPs can be lowered by refining the Bayesian network parameters in such a way that they get closer to the *true* values. In section 2.3.5, a method for automatically updating the beliefs on network parameters was introduced using *data cases*. That is, starting from a set of prior beliefs on the network parameters, the distributions representing these beliefs are updated (incrementally) using vectors of values for each of the random variables in the network. In practice, one can replicate this process by uniformly sampling from the deductions made by the reasoner, and manually verifying whether, for the given sample, an inconsistency or equivalence is present. This leads to values for all random variables, which can then be used to update the distributions. Theoretically, this should decrease the number of FPs.

To verify this hypothesis, 50 sets of models were generated and analyzed. After every generation of a set of models, 10% of the deductions (up to a maximum of 2500) were analyzed by comparing them to the lists of actual equivalences and inconsistencies to generate data cases (individual cases were determined by uniformly sampling from the set of all deductions). These data cases were then stored in an Excel spreadsheet (for later verification and post-processing) and used in updating the distributions on the network parameters incrementally. These updated distributions were then used in reasoning over the model in the successive generation of a set of models. This process is repeated for all 50 sets of models, leading to the generation of $101,019$ data cases. Note that, in the process, the *elicited distributions* were updated incrementally. To better see the impact of learning on the prior distributions, an *equivalent sample size* (see section 2.3) of $100,000$ is used for the prior distributions. This is representative of how *strong* one's belief is [152] (i.e., how large the pool of observed samples is).

The updated network parameters determined after successively learning from 10% of the deductions of 50 sets of generated models were then used in reasoning about inconsistencies and semantic equivalences in 35 newly generated sets of models. This is done to determine averages for recall, precision, specificity, F-measure and incurred cost (depicted in figures 54 and 55).

Immediately noticeable in both plots is an overall increase in precision and specificity. Additionally, it appears as though the trend of the precision curve is becoming

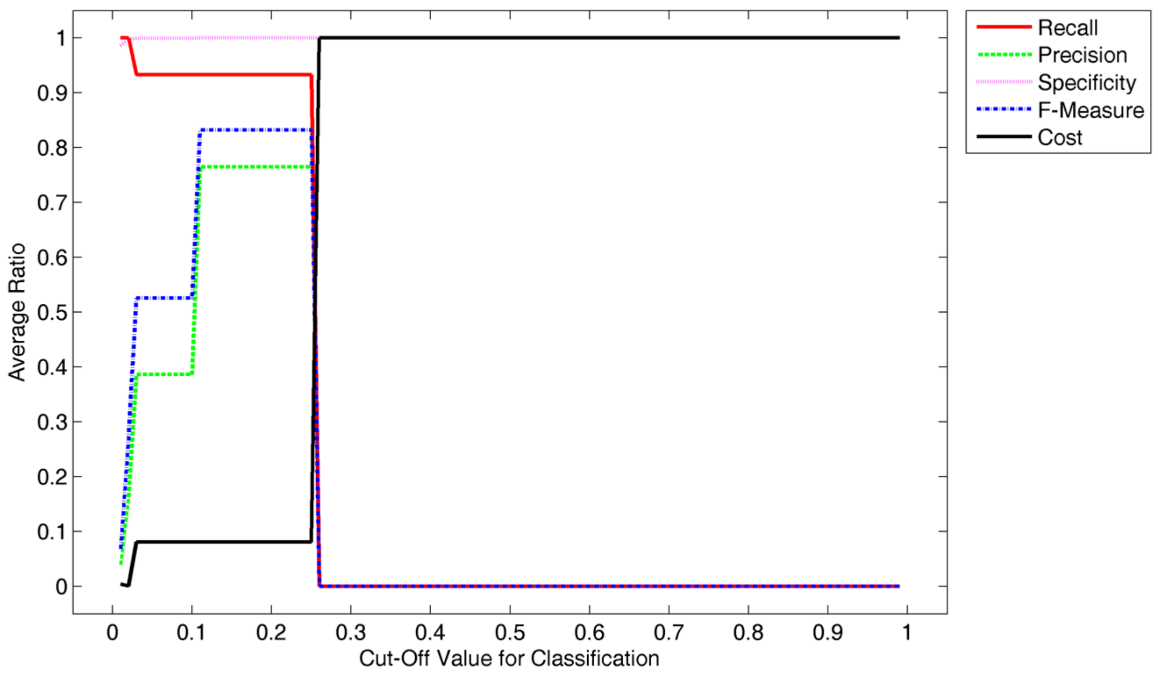


Figure 54: Average recall, precision, specificity, F-measure and incurred cost after learning from 10% of the deductions of 50 sets of generated models (inconsistency classification).

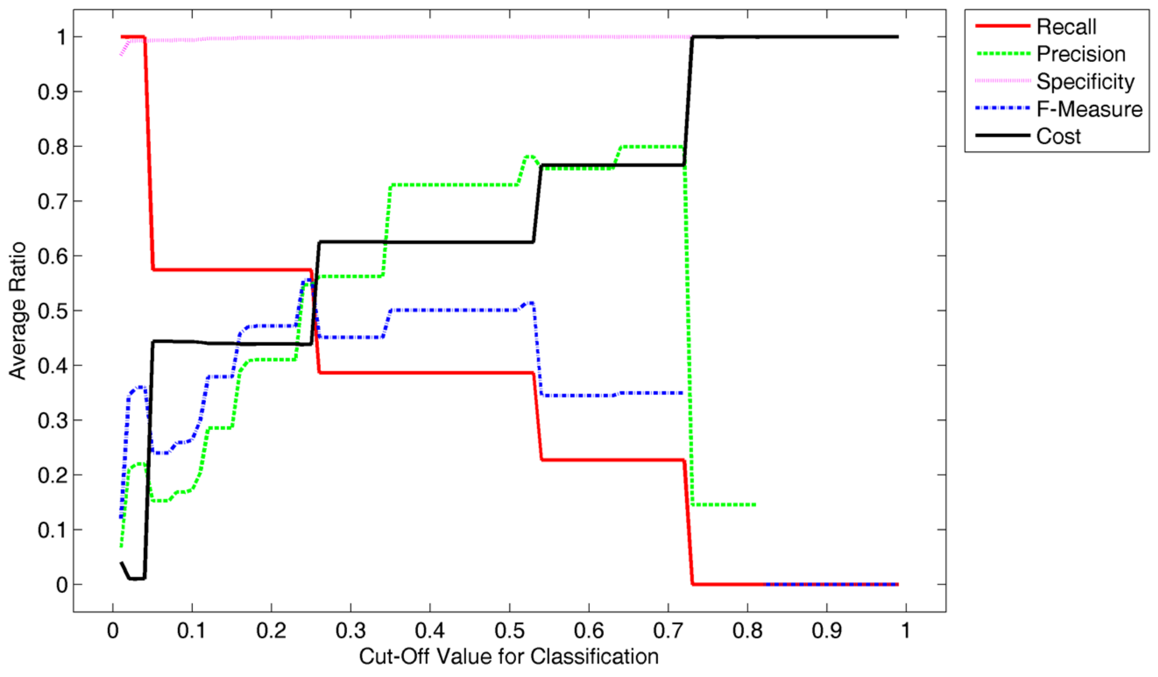


Figure 55: Average recall, precision, specificity, F-measure and incurred cost after learning from 10% of the deductions of 50 sets of generated models (semantic equivalence).

more similar to that observed in figures 51 and 52. However, also noticeable is a decrease in recall for lower cutoff values. It is suspected that this is a result of too few data cases considered to accurately define the network parameters – indeed, comparing the network parameters of the various investigated cases more closely, it appears that some of the *true* distributions are very different than those elicited from a human. This is suspected to be an effect of *regression*: when eliciting the beliefs over the network parameters, the algorithmic injection of inconsistencies and incompletenesses is not considered (although there almost certainly is still a slight bias). For instance, consider the belief on the event *"The parents of the properties are similar with probability 0.8 if the constraints imposed on the properties are exactly the same, and the names of the parent entities have a similarity score value between 0.6 and 0.8"*. This is an extremely rare event and, for the case of the generated models, produces a large number of false positives. Indeed, the *true* probability of this event can be determined analytically to be 0.0072, which is accurate given the specific parameters used for generating models, and the specific circumstances under which inconsistencies and incompletenesses are introduced (i.e., the probability is a result of *fitting* the parameters to the data). However, while *statistically* correct, it does not reflect the belief on *future outcomes* and is outside of the realm of the generated models.

Lastly, what is also very noticeable is a sharp decline in the precision at about 0.71. The curve stops at 0.8, since the value for precision is undefined for the case of no TPs and no FPs (this would lead to 0/0). This coincides with the drop in the recall value (note that the recall *does not fall to* 0 *until after* 0.8 – for the case of no TPs, precision would be 0), which means that the ratio of TPs to FPs is comparatively high. This, once again, is a likely indicator that the Bayesian network and patterns should be refined. Some pairs of properties are recognized as relatively likely to be semantically equivalent, even though they are not in actuality.

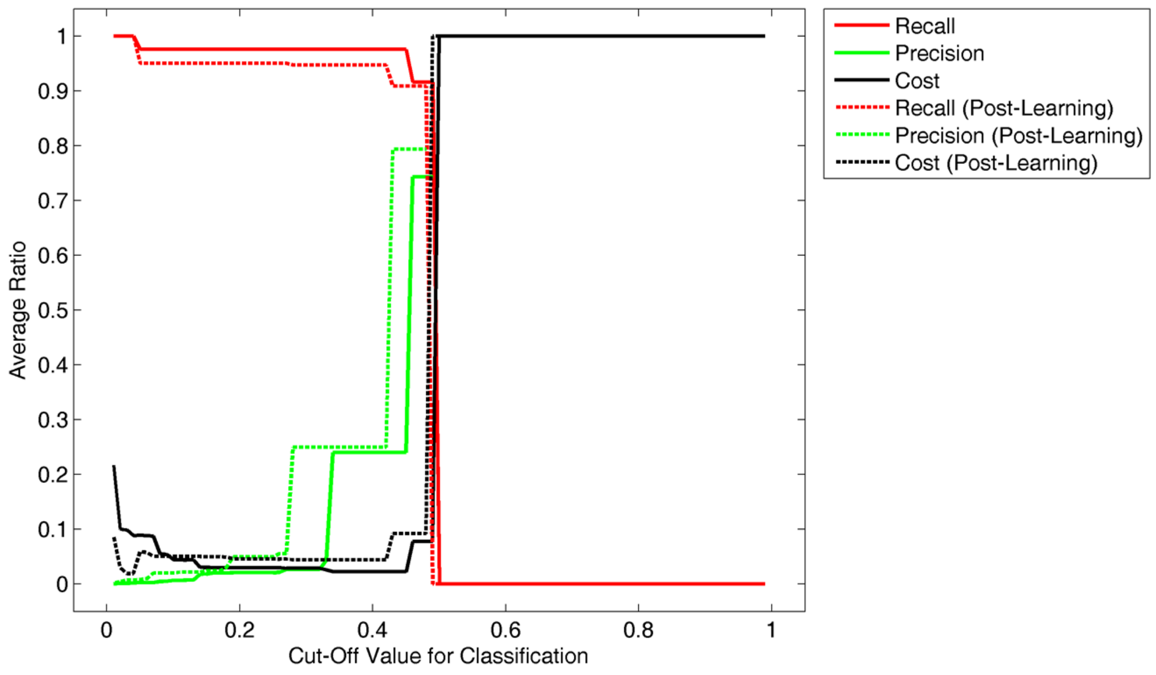


Figure 56: Differences in average recall, precision and incurred cost of the inconsistency classifier after learning from 10% of the deductions of 1 set of generated models.

The experiment is repeated for the case of learning from 10% of the samples of just one set of generated models. As a prior sample size, $2,500$ is used. The resulting smaller set of deductions used as datacases (approximately 2500) represents a more *realistic* case. Given that the prior sample size is of an approximately equally large quantity, the prior is still prevailed. Similar to the previously performed experiment, averages of recall, precision and cost were computed by applying the updated inconsistency identification knowledge to 35 sets of generated models. The results are plotted in figure 56, where the performance is also compared to the results of the initial measurements.

What is immediately visible is a significant reduction in incurred cost (up to approximately 13%) for cutoff values below 0.08. Further investigation reveals that this is due to the reduction in the number of false positives. This can be seen by the shift in the curve of precision towards the trend recognized previously. Note that the

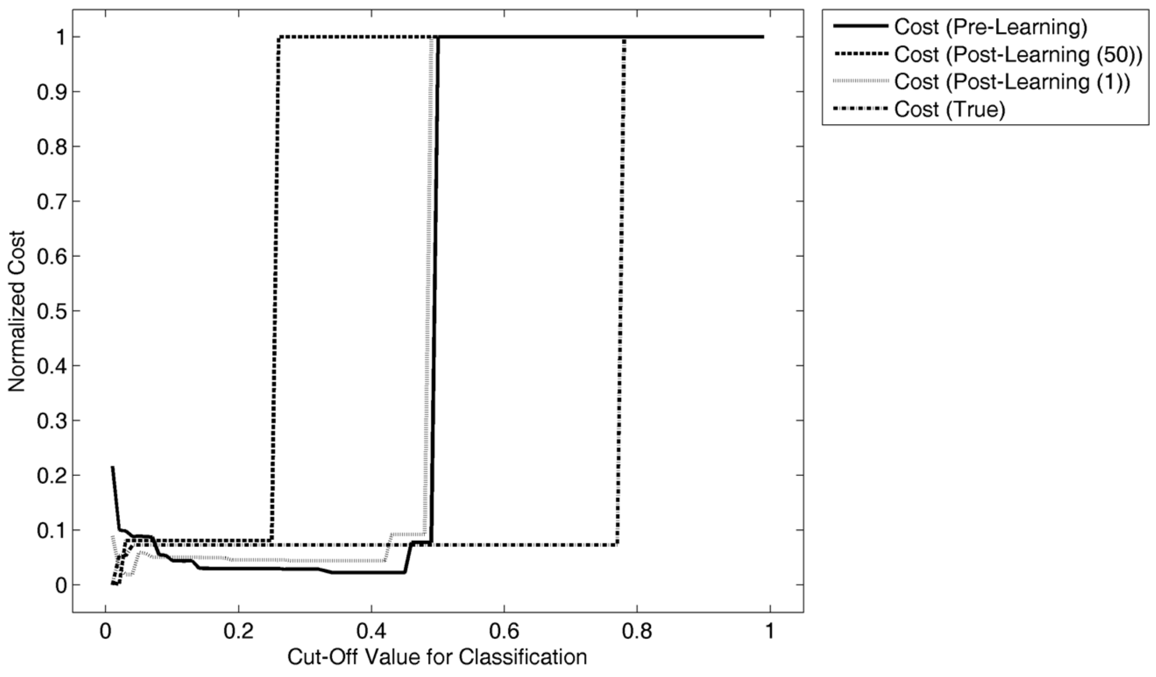


Figure 57: Incurred cost by classification of inconsistencies by the Bayesian network with elicited beliefs on the network parameters compared to the cost incurred using the true frequencies.

slight reduction in recall is a result of a statistical error stemming from the use of different set of generated sets of models for determining the average measures (this is done to emulate learning from previous experience, and applying the results to a similar, but not identical situation).

Finally, the cost incurred after learning is compared to the cost incurred before learning, and to the cost incurred by a classifier that has the *true* network parameters. These various costs are plotted in figure 57. Note that it is valid to compare the (normalized) costs due to the identical conditions under which these curves were created (same model generation parameters, same cost ratios).

Note from figure 57 that, as noted previously, the minimum cost is evidently smaller in both the case of incrementally updating the prior distributions on the network parameters and in the case of using the *true* network parameters. However, similar to the case of using the *true* network parameters, the fact that FNs are produced earlier leads to a higher expected cost for small cutoff values. The fact that the cost incurred by the classifier with subjective beliefs on the network parameters performs better for a wide range of cutoff values as opposed to the other classifiers could be due to two reasons: either a sampling error (i.e., 35 generated models is too few), or simply a result of under-weighing outcomes. The latter is supported by the fact that the recall value is higher, and the precision value lower, in this range for the classifier with subjective beliefs. This means that the classifier is more *conservative* in predicting an inconsistency and is capable of producing more TPs for a wider range of cutoff values. This is, once again, an indication of the robustness of the classifier. It also means that it is sometimes more valuable to produce more false positives in order to avoid false negatives.

Interesting to observe in the case of using large numbers of data cases is the shift of the sudden cost increase cost for lower cutoff values. As discussed previously, this is likely the result of learning with comparatively few data cases. For instance, for the case of the previously mentioned event *"The parents of the properties are similar with probability 0.8 if the constraints imposed on the properties are exactly the same, and the names of the parent entities have a similarity score value between 0.6 and 0.8,"* the probability will decrease as more data cases are considered, while the distributions on other parameters may increase.

8.2.5.6 Impact of Cost Ratio

One of the observations made about the behavior of the classifier in the initial experiment, is that the cost first decreases (and is initially large due to a large number of FPs), and then increases again due to the production of FNs. Interesting to investigate is the impact of C_{ver}/C_{FN}. This ratio is highly dependent on the expected impact of a FN compared to a FP: for instance, in cases where the impact of an undiscovered inconsistency leads to drastic consequences (e.g., mission failure in the case

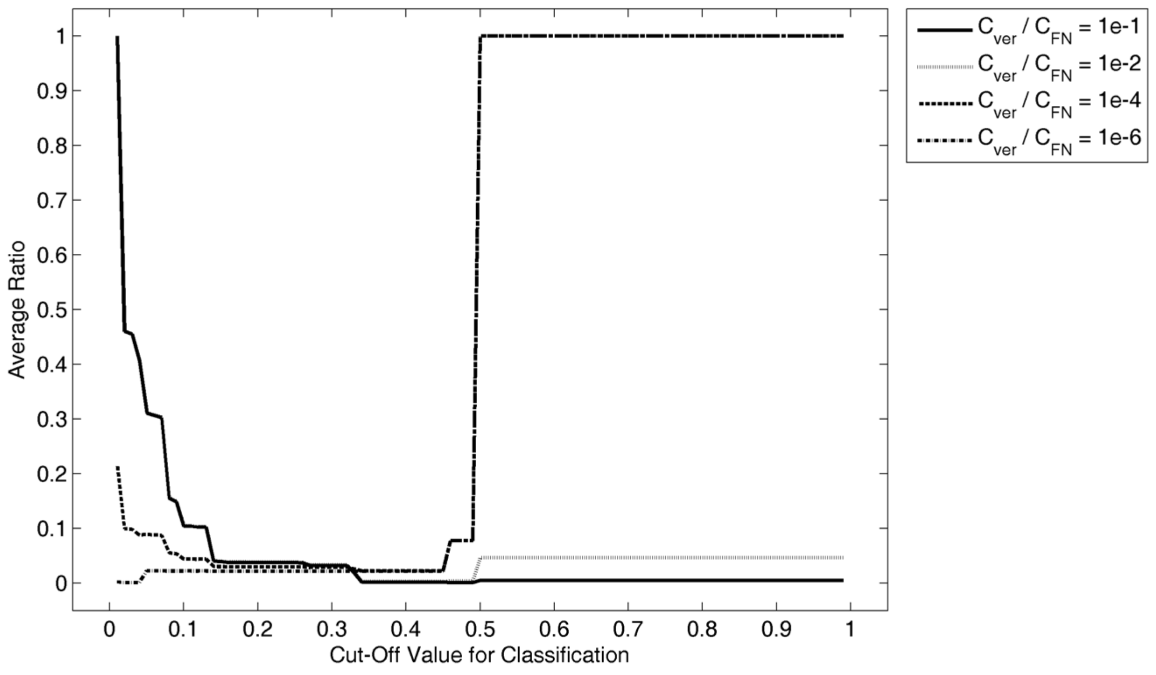


Figure 58: Relative effect of various cost ratios on the incurred cost over the range of cutoff values.

of the MCO (see chapter 1)), this ratio is very small and, hence, $(C_{ver}/C_{FN}) \rightarrow 0$. It is expected that the impact of a FP diminishes in cases of very low ratios, and dominates for small cost ratios.

To investigate this hypothesis, the gathered data (e.g., generated models and deductions made) from the initial experiment are re-used, and the cost incurred under the assumption of various cost ratios are plotted for the case of inconsistency classification. Four ratios are considered: $C_{ver}/C_{FN} = 0.1$, $C_{ver}/C_{FN} = 0.01$, $C_{ver}/C_{FN} = 0.0001$ and $C_{ver}/C_{FN} = 0.000001$. The result of this is illustrated in figure 58. Note that different scaling values are used for each curve (i.e., the maximum actual cost incurred is much higher for the ratio 0.000001 than for 0.01). The aim of plotting the various cost ratios is to illustrate the relative effect of FPs and FNs on the incurred cost.

The non-negligible impact on the incurred cost by the FPs even for relatively low ratios (e.g., 0.0001) is interesting since it shows the importance of carefully having to

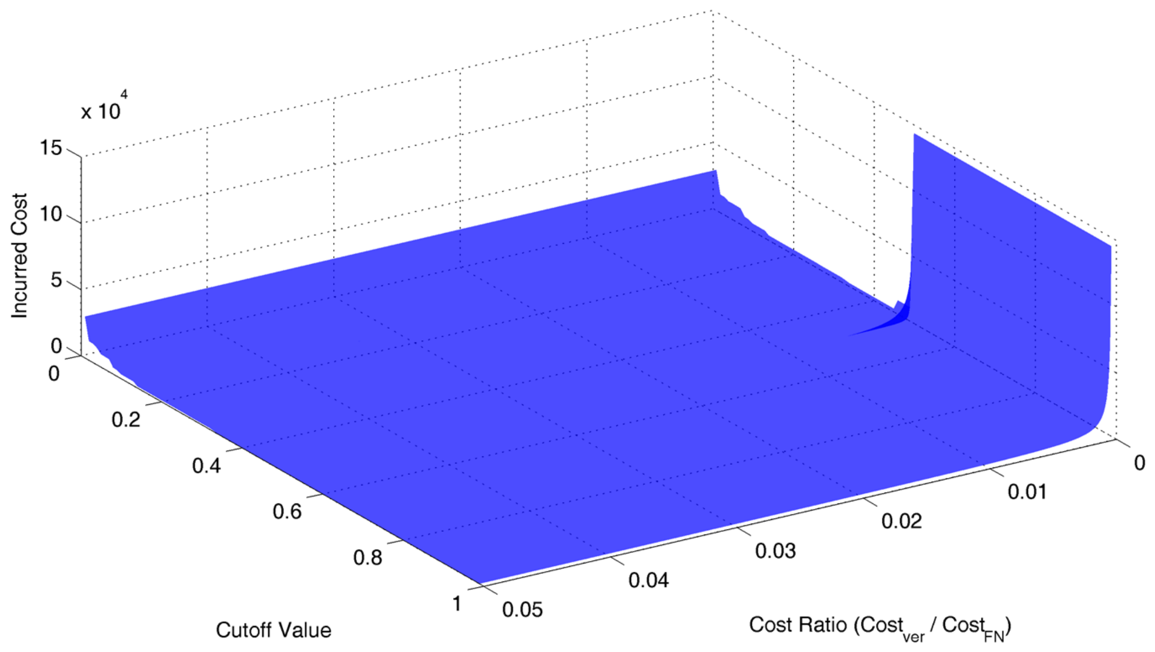


Figure 59: Incurred cost as a function of the cutoff value and cost ratio C_{ver}/C_{FN}.

select cutoff value that is neither too low, nor too high. However, as suspected, for extreme cases (e.g., 0.000001), the *relative* impact of FPs diminishes. Also striking is that the shape of the curve for very small ratios approaches the *shape* of the curve identified for the true network parameters. This is a logical consequence of equation 28. However, important to observe is that, as the relative impact of FNs rises, the *value* of refining network parameters in an effort to reduce FPs likely diminishes.

One additional interesting observation is that a plateau seems to exist in which the cost is comparatively low, irrespective of the cost ratio. This is likely a behavior specific to the classifier (that is, the Bayesian network and patterns). Remembering figure 47, the recall is relatively constant for the same range of probabilities. Given that false negatives have the largest impact, the plateau is no surprise. However, what is striking is that, even for relatively small cost ratios, the impact of FPs is fairly low in this region, even though the precision is very low (and hence the number of false positives is high compared to true positives).

This, combined with the previous observations is indicative of the fact that there is only a relatively small range of cost ratios in which the impact of FNs and FPs are comparable (hence, requiring a trade-off). However, regardless of the cost ratio, there seems to be some range of cutoff values which are always favorable. As mentioned previously, it can be hypothesized that this is likely a behavior specific to the particular classifier used. Figure 59, is a plot of different cost ratios, cutoff values and actual incurred cost (i.e., not normalized). Note the steep incline of the cost as the ratio nears 0 and the cutoff value is large. This is due to the high impact of FNs as compared to FPs. The incurred cost is very small for low cost ratios and high cutoff values due to a low number of FPs and due to an almost negligible effect of the cost incurred by FNs compared to the cost incurred by FPs.

8.2.5.7 Impact of Size of Inconsistency Identification Knowledge

In this section, the effects of using a larger set of inconsistency identification knowledge is investigated. For this purpose, a second, more *comprehensive* Bayesian network is used and the results gathered are compared to the network used in previous experiments. This is done to further investigate the hypothesis that the observed behavior is classifier-specific, and that the plateau is an indicator of humans under-weighing outcomes. In addition, it is determined whether considering more a greater variety of information still leads to the evidence of inconsistency outweighing the evidence of semantic equivalence (see section 8.2.5.2). Models are generated under identical conditions as before, and the same incurred costs by FPs, TPs and FNs are assumed, so that both cases can be compared.

The Bayesian network is illustrated in figure 62. Random variables, target space values, associated patterns and elicited distributions on the network parameters are detailed in appendix A.2. The primary difference from the network used previously is the consideration of the larger semantic context around a pair of properties. For example, the similarity of a property's *range* is considered, as well as the similarity

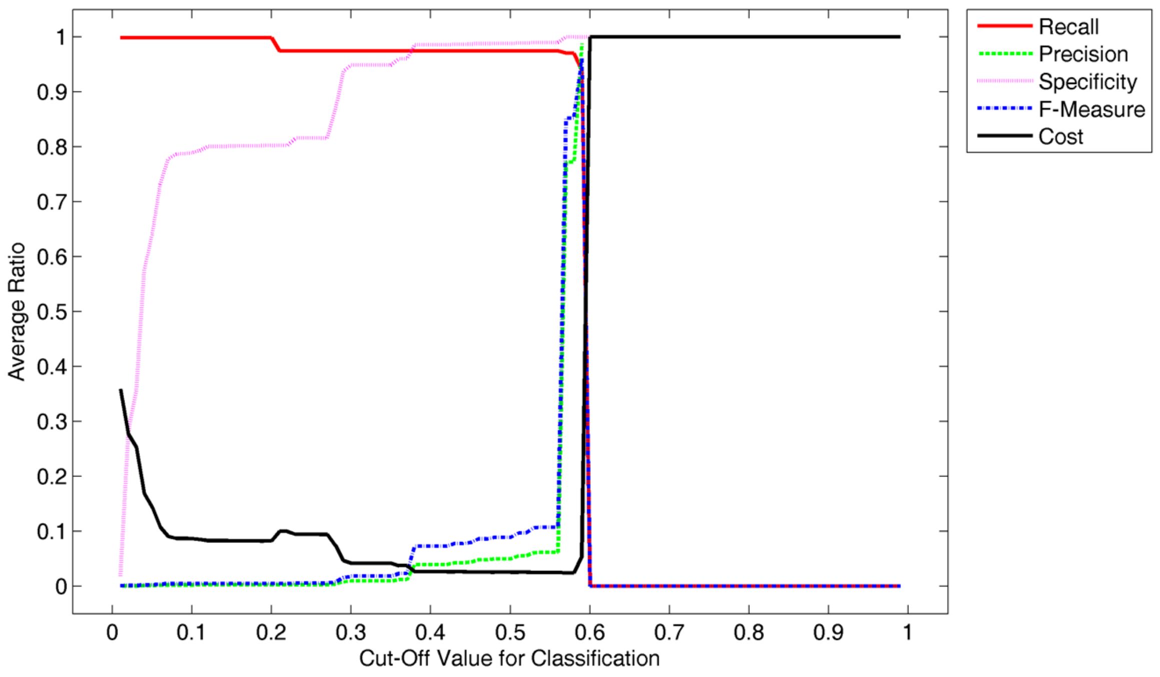


Figure 60: Recall, precision, specificity, F-measure and cost for inconsistency identification averaged over 35 runs and plotted over different cut-off values for classification (comprehensive Bayesian network).

of the types of the owners of the properties. In addition to a similarity score, a check is performed whether the expressions being compared (here: names) are *synonyms*. This check for synonyms is performed by polling the previously introduced WordNet® database. As before, the similarity score defined by equation 29 is used for comparing the similarity of two (textual) expressions. Note that some influence relationships which one may expect to exist in actuality (e.g., between *SameRelationType* and *RelationTypeNamesSimilar*) are left out. This is done to reduce the overall complexity of the network parameters. For parts of the Bayesian network, this leads to conditions similar to those which exist when using a naïve Bayes model (see section 7.2.3.3)).

Figures 60 and 61 depict the averages (again, over 35 generated sets of models) of recall, precision, specificity, F-measure and cost for the case of classifying pairs of properties as inconsistent and as semantically equivalent, respectively. Comparing the plots to the results gathered using the compact Bayesian network (figures 47 and 48),

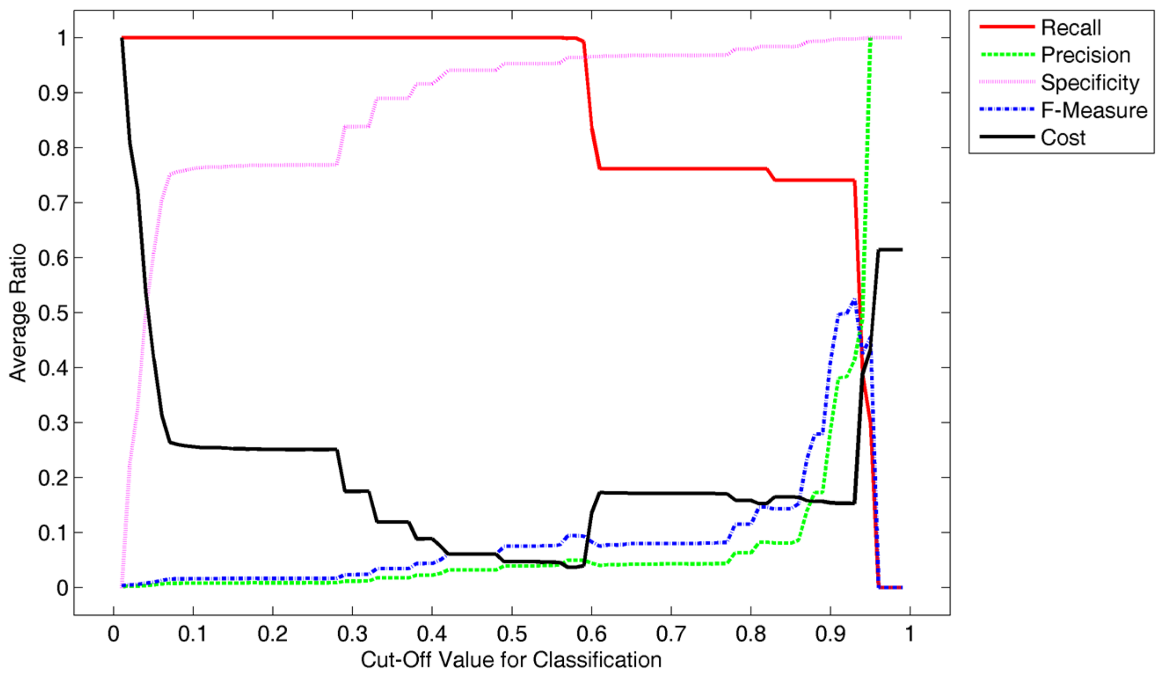


Figure 61: Recall, precision, specificity, F-measure and cost for semantic overlap detection averaged over 35 runs and plotted over different cut-off values for classification (comprehensive Bayesian network).

what immediately becomes evident is the much lower precision over a wide range of cutoff values (and, hence, the much larger impact of FPs on the cost as compared to the FNs). This indicates that a much larger number of false positives are produced as compared to the network that made arguably more naïve assumptions. The reason for this is suspected to be the much larger amount of evidence considered, most of which is not a strong indicator of either inconsistency or semantic equivalence, yet gradually increases the belief. In the case of the inconsistency classifier, this hypothesis is supported by the fact that there is a noticeable increase in precision, followed by a sharp drop in the recall value at around 0.6 (recall that similar observations were made in the case of the compact Bayesian network). The reason for this is that semantic equivalence strongly influences the probability of inconsistency. Sufficient evidence indicating this is suspected to reduce the number of FPs drastically at a cutoff value of 0.56 to 0.58. This is followed by a reduction in the recall value, which indicates

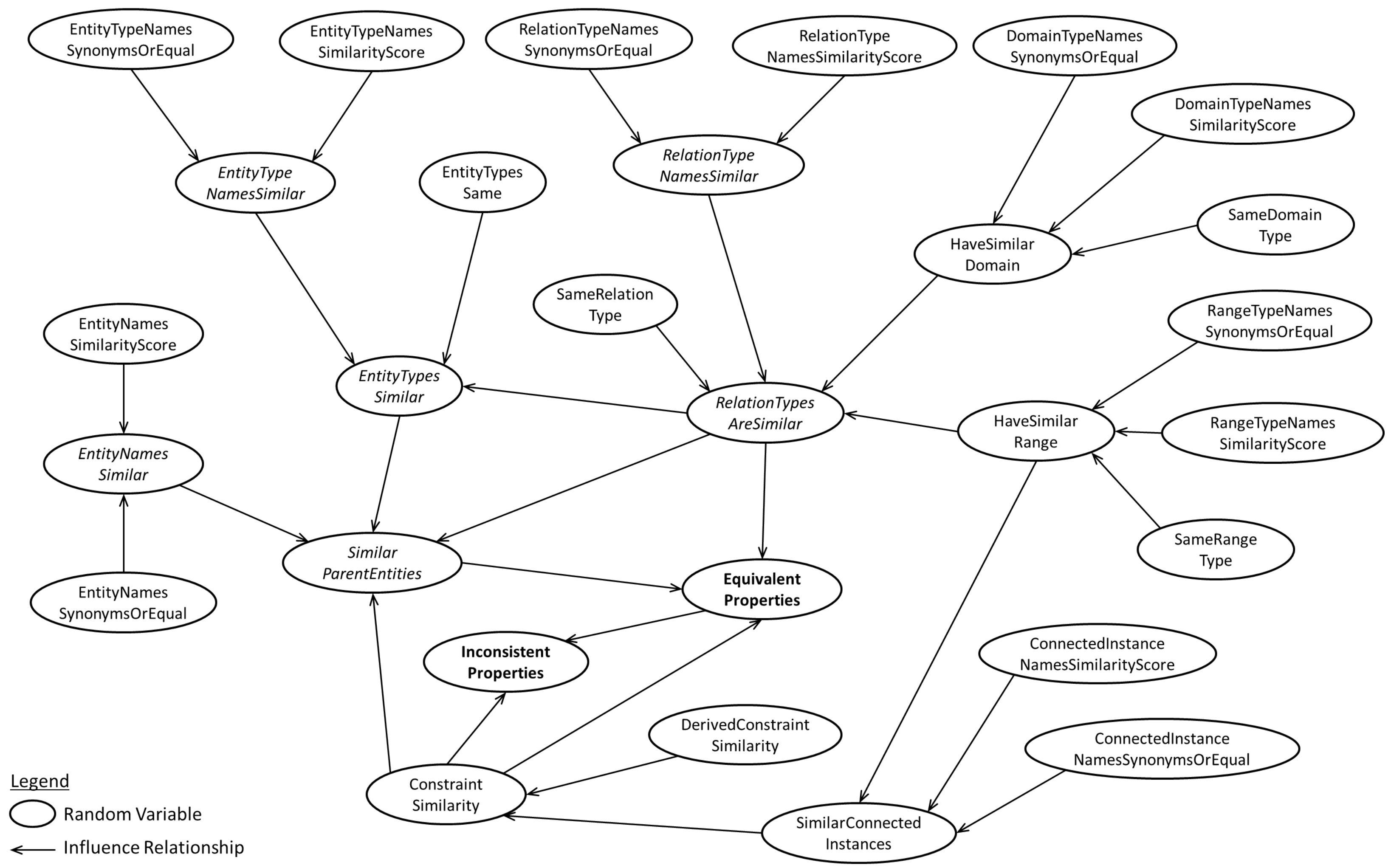

EntityTypeNames SynonymsOrEqual
EntityTypeNames SimilarityScore
RelationTypeNames SynonymsOrEqual
RelationType NamesSimilarityScore
DomainTypeNames SynonymsOrEqual
DomainTypeNames SimilarityScore
EntityType NamesSimilar
EntityTypes Same
RelationType NamesSimilar
SameDomain Type
HaveSimilar Domain
SameRelation Type
RangeTypeNames SynonymsOrEqual
EntityNames SimilarityScore
EntityTypes Similar
RelationTypes AreSimilar
HaveSimilar Range
RangeTypeNames SimilarityScore
EntityNames Similar
SameRange Type
Similar ParentEntities
Equivalent Properties
EntityNames SynonymsOrEqual
ConnectedInstance NamesSimilarityScore
Inconsistent Properties
DerivedConstraint Similarity
ConnectedInstance NamesSynonymsOrEqual
Legend
Random Variable
Influence Relationship
Constraint Similarity
SimilarConnected Instances

Figure 62: Comprehensive Bayesian network used in reasoning about the semantic overlap and inconsistency of distinct pairs of properties by analyzing their larger context.

that at that point (similar to before) evidence that the properties are semantically different (in some sense inconsistent) outweighs evidence that they are (intended to be) equivalent. This is, once again, supported by the sharp drop in the recall value for the semantic equivalence classifier (figure 61), which indicates that those pairs of properties that are intended to be semantically equivalent, but are also inconsistent, are no longer classified as inconsistent or semantically equivalent (due to these being considered *too different* by either classifier).

One additional interesting observation is a slight decrease in the recall value at 0.2 for identifying inconsistencies, but not semantic equivalences. It is suspected that this is due to a relatively rare, but nonetheless possible case that can be introduced when injecting the generated models with inconsistencies. As outlined in section 8.2.2.4, it is possible for the length of two semantically equivalent segments to have the same value, but different *intended* units – that is, no units are specified, but the modelers intended to use different units. Such cases are, of course, almost impossible to identify without further information, or by choosing a low cutoff value (here, a cutoff value below 0.2) and accepting the production of a large number of FPs.

An effect of the (comparatively) very low precision over a wide range of cutoff values is also a lower specificity over this range. As before, this is indicative of a need to revise the inconsistency identification knowledge and, in particular, the distributions imposed over the network parameters. Such is (by inference from previous observations) expected to reduce the number of false positives.

Even though the precision is evidently low for a wide range of cutoff values, plotting the ROC curve for the inconsistency classifier still indicates a very good performance of the classifier. This is striking, since the cost curve seems to suggest the necessity of refining the classifier. Also note the maxima for the F-measure: in the case of the inconsistency classification (figure 60), the F-measure is at a maximum when the cost is also at a minimum. However, for the equivalence classification, the

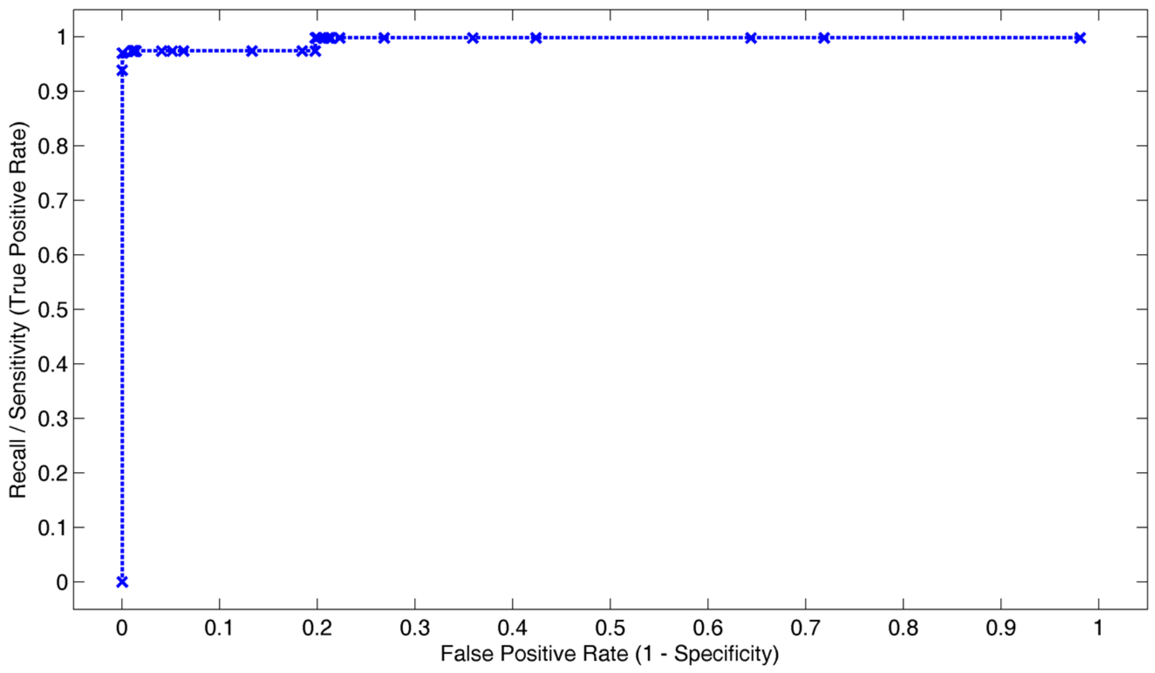


Figure 63: Receiver operating characteristic (ROC) for inconsistency identification averaged over 35 runs (comprehensive Bayesian network).

F-measure suggests a much higher cutoff value (≈ 0.93) than does the cost curve (≈ 0.58). This indicates that – at least for the case of inconsistency identification – the commonly used F-measure is not a very good metric to be used for identifying an adequate value for the classification cutoff value, but that a value-based metric is better on average. This also avoids having to artificially *tune* the parameter β of the F-measure (to put more weight on either recall or precision) for producing results comparable to the value-based measure.

To better illustrate the impact on the costs, those incurred by the inconsistency classifier and semantic equivalence classifier are depicted in figures 64 and 65 respectively, where they are compared to the cost curve of the compact Bayesian network. Note that in producing the curves, it is assumed that the cost of the test C_{test} is *equal* in both cases. This is a strong assumption, given the larger number of possible states across the comprehensive Bayesian network and the associated increase in computational cost for pattern matching. However, from a practical point of view, the cost

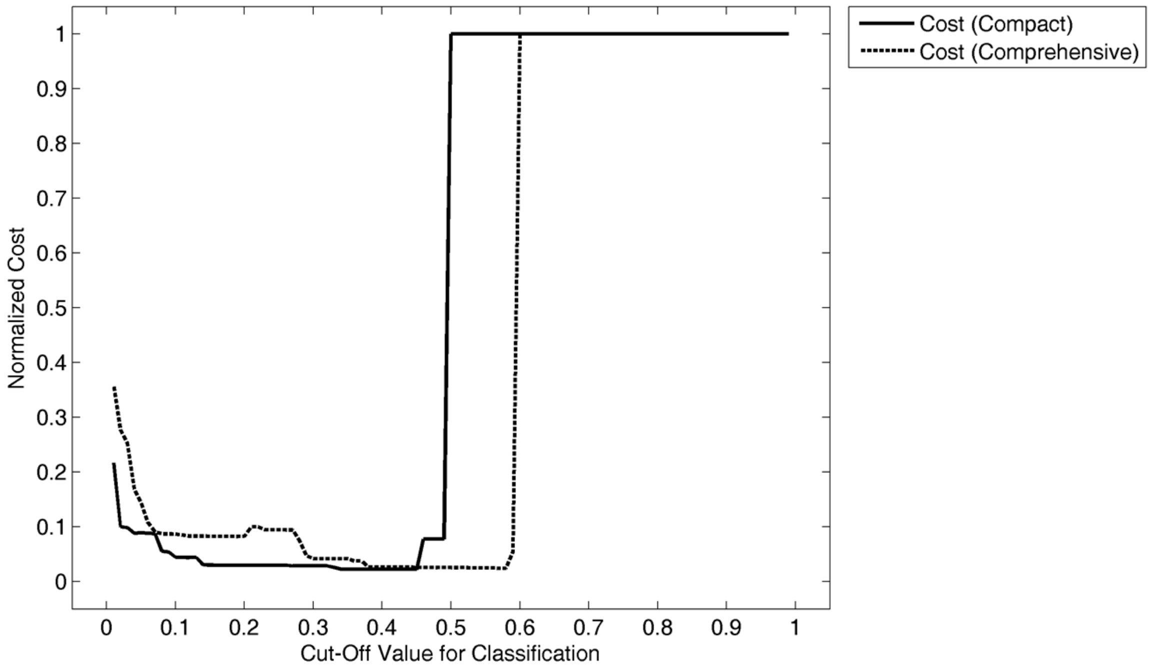


Figure 64: Comparison of cost incurred by the inconsistency classifiers (compact and comprehensive Bayesian network).

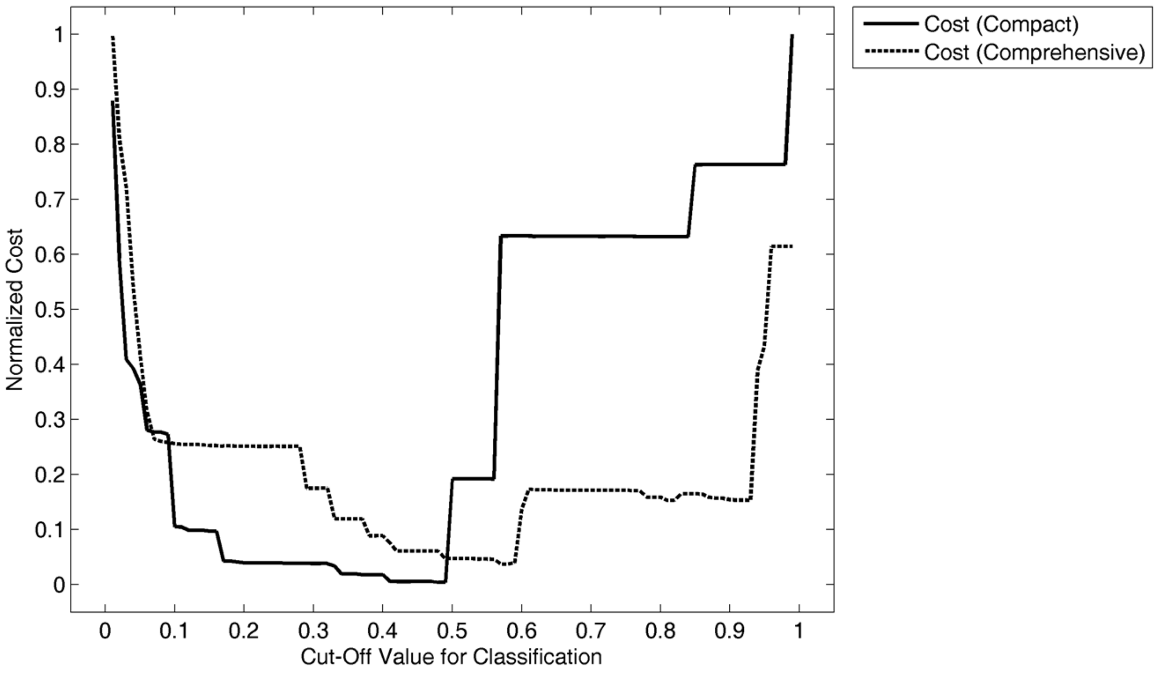


Figure 65: Comparison of cost incurred by the semantic equivalence classifiers (compact and comprehensive Bayesian network).

is negligible with respect to the relatively modest size of the models considered and in relation to the cost incurred by verification of TPs and FPs, and the cost incurred by FNs. Whether this is a generally valid assumption is investigated in a subsequent experiment (see section 8.3.2).

Notice that in the cost plots depicted by figures 64 and 65, the minimum cost incurred by the comprehensive Bayesian network classifier(s) is slightly higher than that incurred by the compact classifier(s). However, the consideration of additional evidence formed a *plateau* wider than that of the compact classifier(s). That is, because of the additional evidence considered, the sensitivity of incurred cost with respect to choosing a cutoff value is noticeably smaller for a wide range of cutoff values. This is because the impact of individual observations that act as weak evidence in support or opposition of inconsistency or equivalence is generally smaller. However, this does not mean that considering more evidence is always better, since the cost incurred by setting up and maintaining the inconsistency identification knowledge is expectedly much larger. Also, the cost of a test is a significant consideration. This results in a trade-off that must be considered.

8.2.6 Comparison to Deterministic Case

One of the conclusions reached in the previous section is that the initial and maintenance cost, as well as computational cost, associated with the application of the approach influences the size of the inconsistency identification knowledge and, specifically, the size of the Bayesian network and pattern complexity. However, one question that has not yet been investigated is whether the proposed method outperforms a state-of-the-art deterministic, pattern- (or negative constraint-) based approach. This question is investigated in this section.

To compare the approach as best as possible, a pattern was constructed that makes similar assumptions to the (compact) Bayesian network utilized in the last section. That is, semantic equivalence is based on representation conventions, where properties are deemed semantically equivalent if their names, and the names of their owners match. This is similar to the assumptions made by the fully automated approaches described in section 3.3. To ensure comparability, the pattern was queried over *the same set of models generated during the initial experiment* (section 8.2.5.2). The patterns (and queries) used are written as SPARQL 1.1 [226] compliant queries, and are made available in appendix A.3.

Queries were written for both identifying inconsistencies and semantic equivalences, and for determining the number of TPs and FPs. The quantity of FNs is determined by subtracting the number of TPs from the number of actual inconsistencies or semantic equivalences. Note that the number of TNs were not computed. The number of TPs, FPs and FNs were determined for all 35 generated sets of models and their averages computed. From these, recall and precision are calculated and the averages for inconsistency identification and semantic overlap detection are depicted in figures 66 and 67 respectively. There the results of the deterministic case are overlaid with the results from the initial experiment.

Note the relatively high precision of the deterministic classifier for inconsistencies. This is due to the fact that *all* (base) properties are considered, including those with non-numeric values. For instance, semantically equivalent `Segment`s may have semantically equivalent *connectsTo* attributes. However, since the range of these properties is non-numeric, the connected instance is not comparable by simple value equality. Interesting to observe about the deterministic inconsistency classifier is that it is *impossible* for it to identify all cases. While the recall is comparatively high, it is, unlike the Bayesian classifier, not capable of identifying all (or more) inconsistencies at a slightly higher incurred cost.

For the case of semantic overlap detection, the precision of the deterministic classifier is fairly high. This is because, unlike in the case of the Bayesian network, the

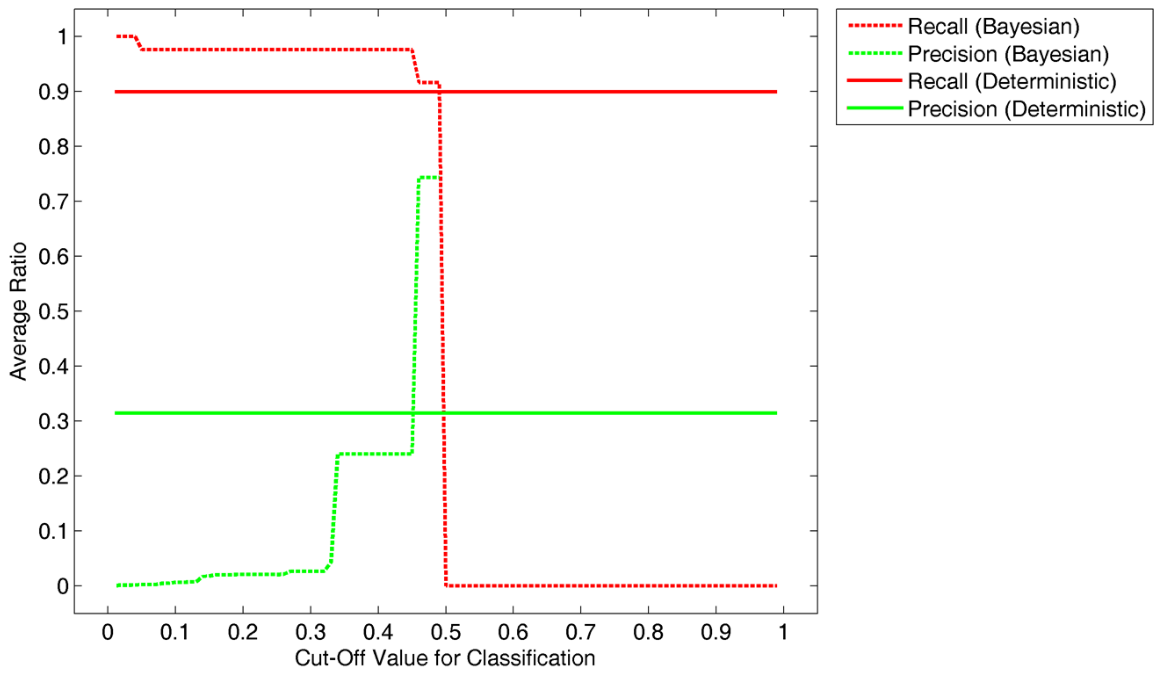


Figure 66: Comparison of recall and precision of a deterministic classifier vs. the proposed Bayesian classifier (illustrated for the case of inconsistency identification).

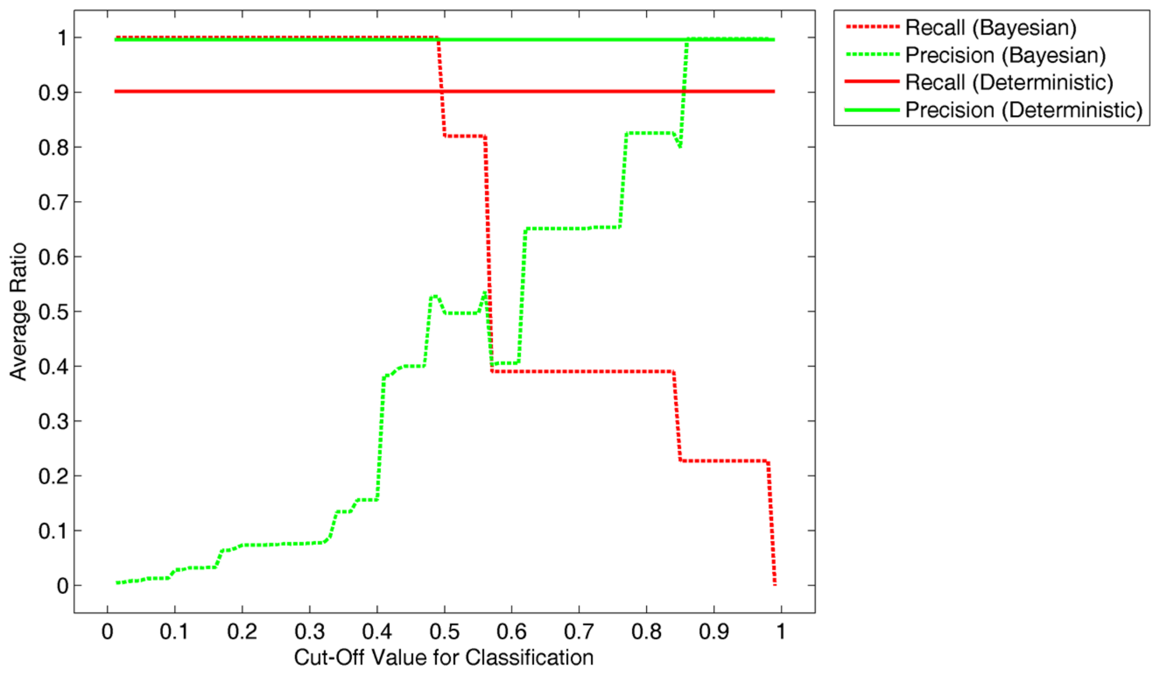


Figure 67: Comparison of recall and precision of a deterministic classifier vs. the proposed Bayesian classifier (illustrated for the case of semantic overlap detection).

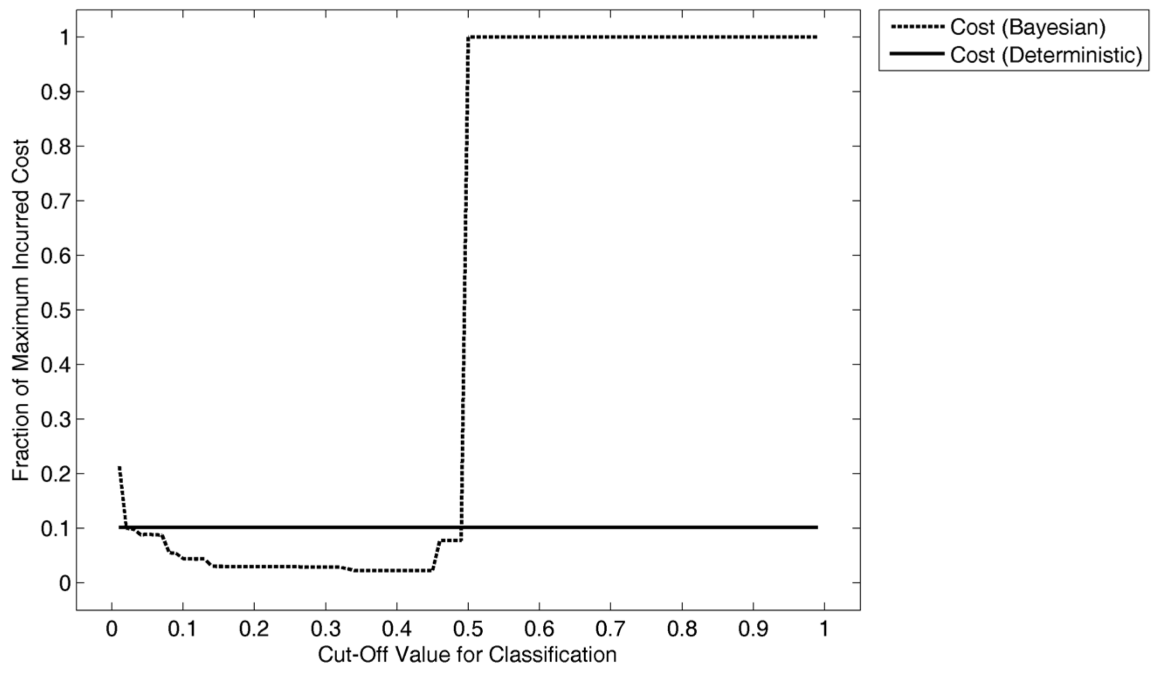


Figure 68: Comparison of value (based on cost incurred) of a deterministic classifier vs. the proposed Bayesian classifier (illustrated for the case of inconsistency identification).

values assigned to the pair of properties are *not* compared and have no influence. Also, it is very unlikely for two non-equivalent properties with equal names, whose parents also have the same name to be generated. Note that the recall is also fairly highly but, similar to the case of inconsistency identification, instances where synonyms are used, or spelling mistakes introduced (along with other such modifications) are impossible to detect by the pattern. Therefore, unlike the Bayesian classifier, the deterministic classifier is incapable of identifying *all* semantically overlapping pairs of properties.

To better compare the deterministic and probabilistic case, consider figures 68 and 69 which depict the cost incurred by either type of classifier, given the same cost ratio as used throughout the previous experiments. Note that the cost of the test is not included, and setup and maintenance costs are not factored. Judging by these results, and comparing them to previous comparisons of classifier cost (see, e.g., figure 53),

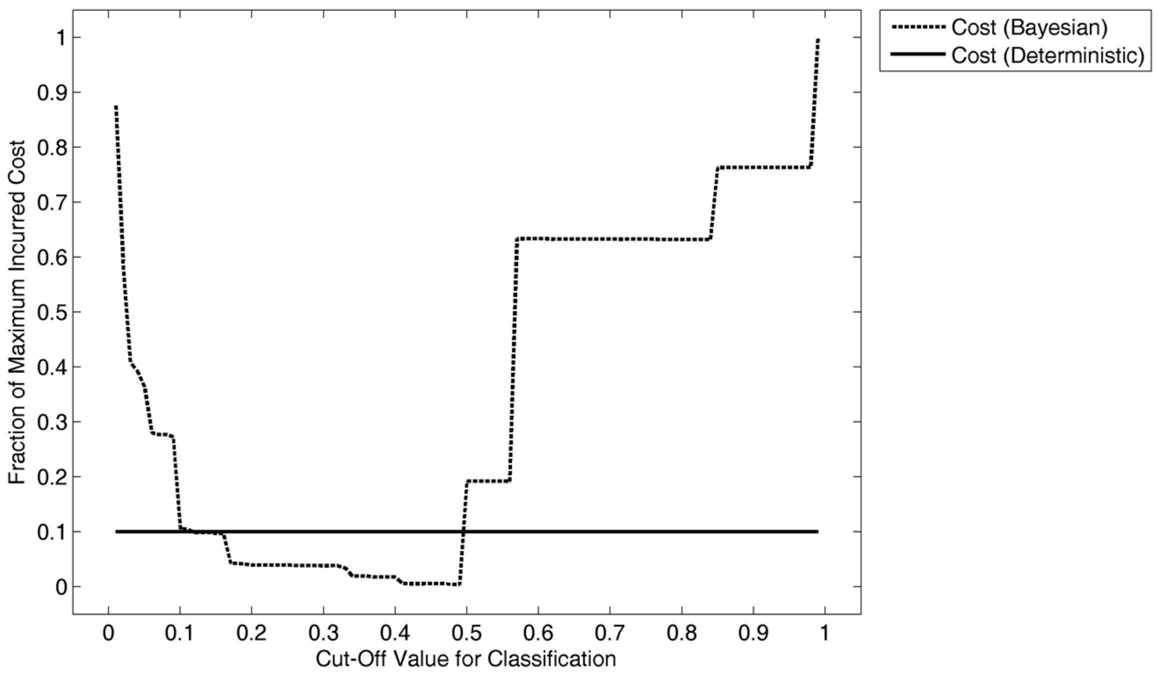


Figure 69: Comparison of value (based on cost incurred) of a deterministic classifier vs. the proposed Bayesian classifier (illustrated for the case of semantic overlap detection).

it appears that for certain cutoff values the Bayesian classifier invokes less cost while achieving a higher precision and recall. For instance, consider the cutoff range of 0.18 to 0.48: there, the cost of either Bayesian classifier is lower (and, hence, its expected value higher) than that of the deterministic classifier. In the same range, the recall of the Bayesian classifier is also *higher* than that of the deterministic classifier (at the expense of lower precision). Note that this is the case for even this relatively *simple* scenario, where the simulated degree of inconsistency and incompleteness is relatively moderate.

The observations suggest the following: compared to a deterministic classifier, using a Bayesian classifier *can* be more preferred. However, this depends on how important it is to be able to identify *all* inconsistencies (e.g., how large the impact of FNs is on the incurred cost), and is, therefore, strongly related to the system under development. In conjunction, the preferred classifier also depends on *how inconsistent* the underlying models are expected to be. A Bayesian classifier is capable of providing useful results with partial evidence, while a deterministic classifier requires the full antecedent to act as a sufficient condition for an inconsistency. This would expectedly lead to a decrease in recall (and, potentially, precision) as the degree of inconsistency of the models increases. Additionally, compared to a Bayesian classifier, using a deterministic classifier has the disadvantage that very large, complex patterns need to be created and maintained, and that separate patterns are required for related reasoning tasks. For instance, for performing the experiments in this section, separate patterns had to be created for semantic overlap detection and inconsistency identification (which embedded the pattern for semantic overlap detection) (see the patterns in appendix A.3). However, only one Bayesian network had to be created for the same task. In practice, it is likely that *variants* of these patterns will be created to account for special cases. This is a common issue with deterministic classifiers which lead to higher expected maintenance costs. This is a known issue with, e.g., rule-based spam filters [5]. Expectedly, this maintenance cost is smaller for a comparably expressive Bayesian classifier.

8.3 Algorithm Evaluation

In this section, the complexity of algorithms 1 and 2 introduced in section 6.3.1 is evaluated. This is done from a standpoint of theoretical complexity, and through empirical performance measurements.

8.3.1 Complexity Analysis

To determine the theoretical complexity of the algorithms, the *uniform* cost model is used [146]. In the following pages, $\tau_{j,i}$ denotes the number of *time units* used for executing line i of algorithm j. In general, it is assumed that atomic operations, such as equality checks and assignments have a worst-case runtime of $O(1)$ (that is, for these operations $\tau = 1$). *Note that, in the following, the behavior of the algorithm with respect to the number of triples being reasoned over is analyzed.*

Let t be the number of triples added and t_G the number of triples in the data graph. Furthermore, let r be the number of random variables in the Bayesian network, v the largest number of target space values among all random variables, and p the number of clauses (triples and functors) in the largest pattern associated with the Bayesian network. Additionally, let o denote the number of possible outcomes in the data graph and o_b the largest number of observations made about any outcome. The complexity in terms of time units of algorithm 1 can then be expressed as:

$$t(r(\tau_{1,4})) + \tau_{1,7} + o(o_b(\sum_{k=10}^{13} \tau_{1,k}) + \tau_{1,15} + (r-1)(\sum_{l=17}^{20} \tau_{1,l})) \tag{30}$$

Similarly, the complexity of algorithm 2 can be determined to be:

$$v(\tau_{2,3} + \tau_{2,4} + \tau_{2,6} + o_b(\tau_{2,8} + \tau_{2,9} + \tau_{2,10})) \tag{31}$$

For simplicity, it is assumed that $\tau_{1,k} = 1$ for $k = 7, 10, 11, 12, 13, 15, 17, 18, 19, 20$ and $\tau_{2,l} = 1$ for $l = 3, 8, 9, 10$. Note that this is a strong assumption for most of these lines (except $k = 7, 11$ and $l = 3, 9$). For instance, for the union and difference operations, the runtime is closer to $n\log(n)$ (given a hashing strategy) with n denoting the size of the list, since lookups are performed rather than just adding values to a list. However, for practical cases, it is deemed negligible compared to other operations involved. Furthermore, for probabilistic inference in the Bayesian network, this assumption is also very strong. However, the runtime of the junction tree algorithm is independent of the number of triples added to the graph, and can be considered negligible compared to the pattern matching operations for most practical situations (i.e., very large data graphs and comparatively small Bayesian networks). Assuming $\tau_{1,4}$ is purely dictated by the runtime of algorithm 2, and making use of the assumptions stated previously, equations 30 and 31 reduce to the following single expression:

$$t(r(v(1 + \tau_{2,4} + \tau_{2,6} + o_b(1+1+1)))) + 1 + o(o_b(4) + 1 + (r-1)(4))$$

$\tau_{2,4}$ involves a lookup in the list of triples t_G. Therefore, one can say that the operation is $O(t)$. $\tau_{2,6}$ involves matching clauses against the data graph. Assuming a comparison of each clause in the pattern to each triple in the data graph, this operation is $O(p \cdot t)$. Assuming the worst case of $t = t_G$, the (approximate) algorithm complexity can be said to be $O(t^2 r(v+p) + to_b 3p + 1 + o(o_b + 1 + 4(r-1))$ under the given assumptions. Finally, assuming a constant Bayesian network (with associated patterns), the simplified complexity is then $O(c_1 t^2 + c_2 t + c_3)$ or simply:

$$O(t^2) \tag{32}$$

8.3.2 Empirical Performance Measurements

To verify the *actual* performance and compare it to the theoretical performance determined in the previous section, CPU timing measurements collected while performing the experiments from section 8.2.5 are presented in this section. In addition, the peak memory consumption of the algorithm is analyzed.

Figure 70 depicts the *CPU time* required for the probabilistic inexact reasoning engine to exhaustively reason about all possible outcomes over the full data graph for 123 generated models. The results from the data collected in the experiment described in section 8.2.5.4 are used for this purpose. The figure depicts the total CPU time required (in seconds), which is plotted against the number of triples in the graph being reasoned over (here: the data graph with all mediations applied). Note that the CPU time is the sum of the *user time* and *system time*. User time is the time spent to run an application's code, and system time is the time spent by the processing unit to run OS code on behalf of the application (e.g., disk input/output)[6]. Note that there is some *spread* of data points, which is particularly noticeable as the models grow in size. This is an effect of the built-in, non-deterministic garbage collection process

[6]Measuring system and user time is *not* simply the difference in two timer values. These more sophisticated tools for timing are available since Java 5 (Java 1.5).

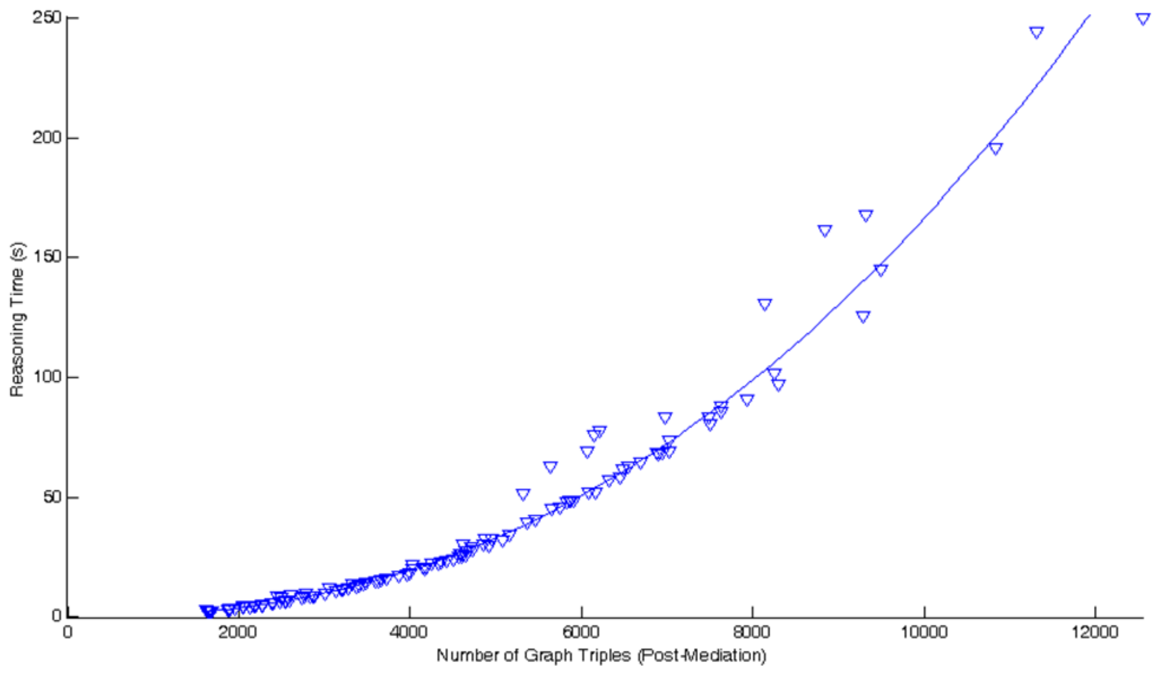


Figure 70: Measured CPU time utilized during the execution of the Bayesian reasoner, plotted against the number of graph triples in the data graph (with mediation rules applied exhaustively, and all RDF/RDFS inferences applied).

executed by the Java virtual machine at non-controllable points in time.

To verify whether the data exerts the trend suggested in the previous section ($O(t^2)$), a log-log plot can be constructed. This is illustrated in figure 71. In a log-log plot, both the measured CPU time and number of triples reasoned over are scaled logarithmically. If the resulting graph follows the expected trend of $O(t^2)$ the slope should then be ≈ 2 (note that the use of "approximately" is useful here due to the strong assumptions made when deriving the theoretical complexity). Inspecting the plot visually, this seems to be the case.

Using the regression tools for fitting curves to scatter plots from Microsoft Excel 2010, the best fitting equation (with $R^2 = 0.992$) was determined to be $y_{power} = 8 \cdot 10^{-8} x^{2.3297}$ (where y refers to the y-axis (i.e., reasoning time) and x refers to the number of triples. A polynomial with order 2 has a similarly good fit with $R^2 = 0.9758$. The polynomial determined is $y_{poly} = 2 \cdot 10^{-6} x^2 - 0.0036x + 2.5245$.

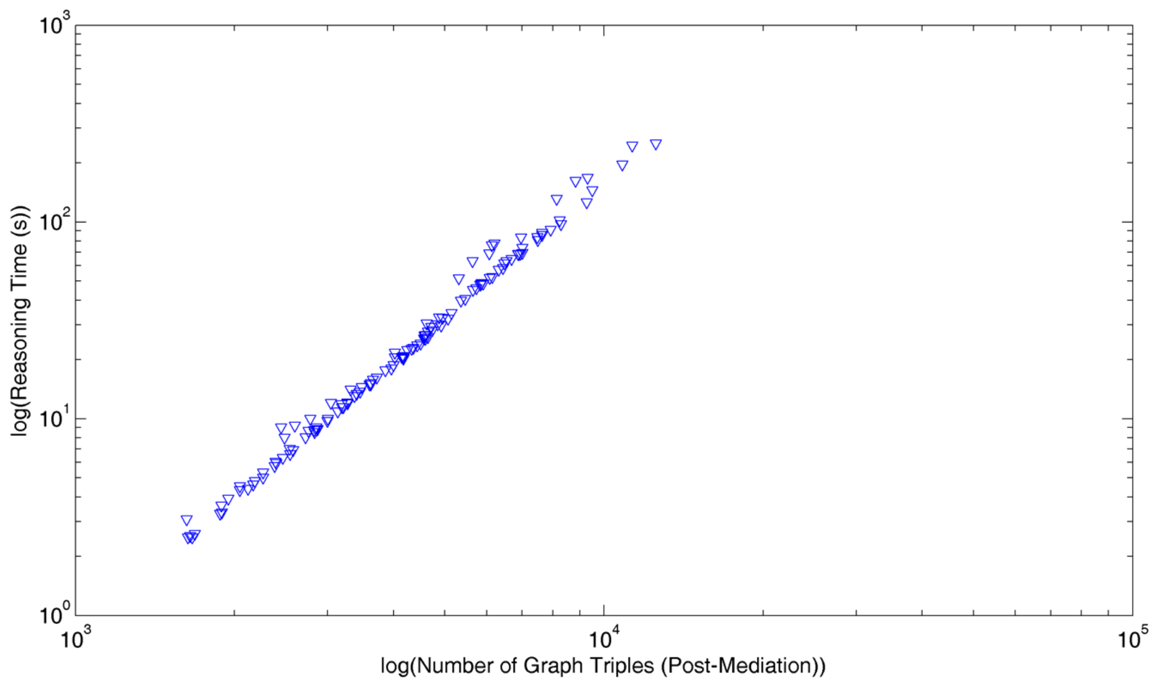


Figure 71: Measured CPU time utilized during the execution of the Bayesian reasoner (log-log plot).

This coincides well with the expected results.

Since a second Bayesian network was utilized during the experiments, results from reasoning using this (considerably more complex) Bayesian network are plotted in figure 72. Note that only 35 data points as opposed to the previously used 123 data points were available. The determined best fitting curve (with $R^2 = 0.9974$) is $y = 2 \cdot 10^{-6} \cdot x^{2.3494}$. Note that the power values are very close in both cases. Indeed, their difference is less than 1% which is, considering that fewer data points were available in the second case, a very close match. This indicates a mere change in the multiplicative constant if different Bayesian networks are used over similarly sized models, such as is predicted by the theoretical complexity analysis (see equation 32).

Note that the mediation time is negligible compared to the time required for the probabilistic inexact reasoning process to fully execute. The CPU times required for exhaustively applying all mediation rules to 123 generated models is illustrated in figure 73. This illustrates the considerably higher computational complexity of the

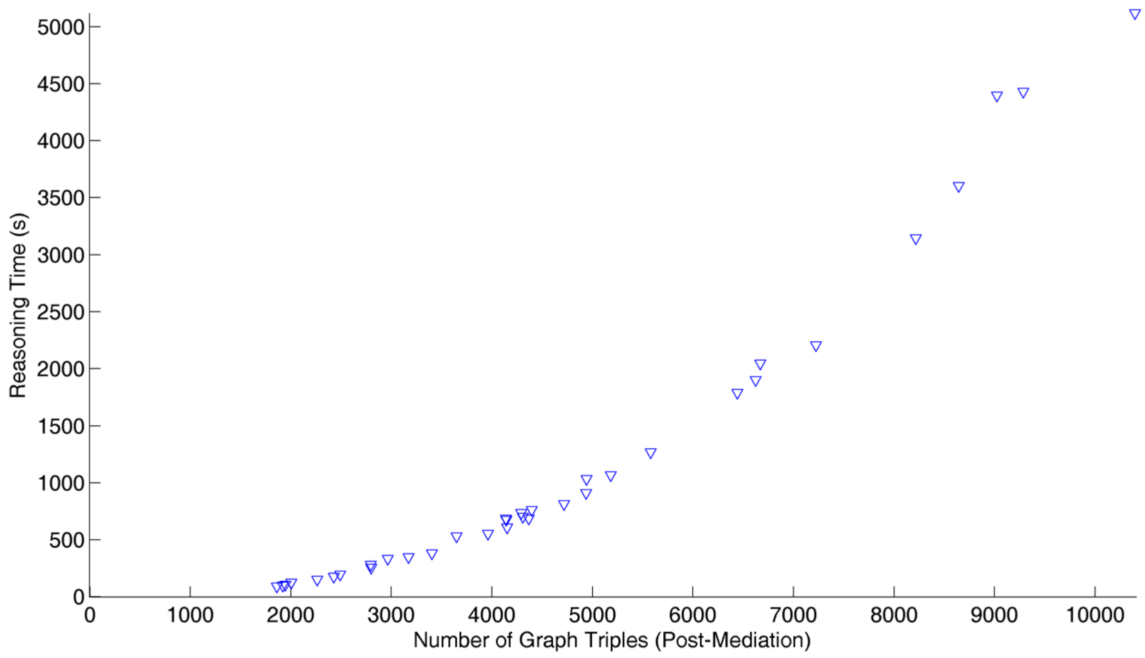


Figure 72: Measured CPU time utilized during the execution of the Bayesian reasoner (comprehensive Bayesian network), plotted against the number of graph triples in the data graph (with mediation rules applied exhaustively, and all RDF/RDFS inferences applied).

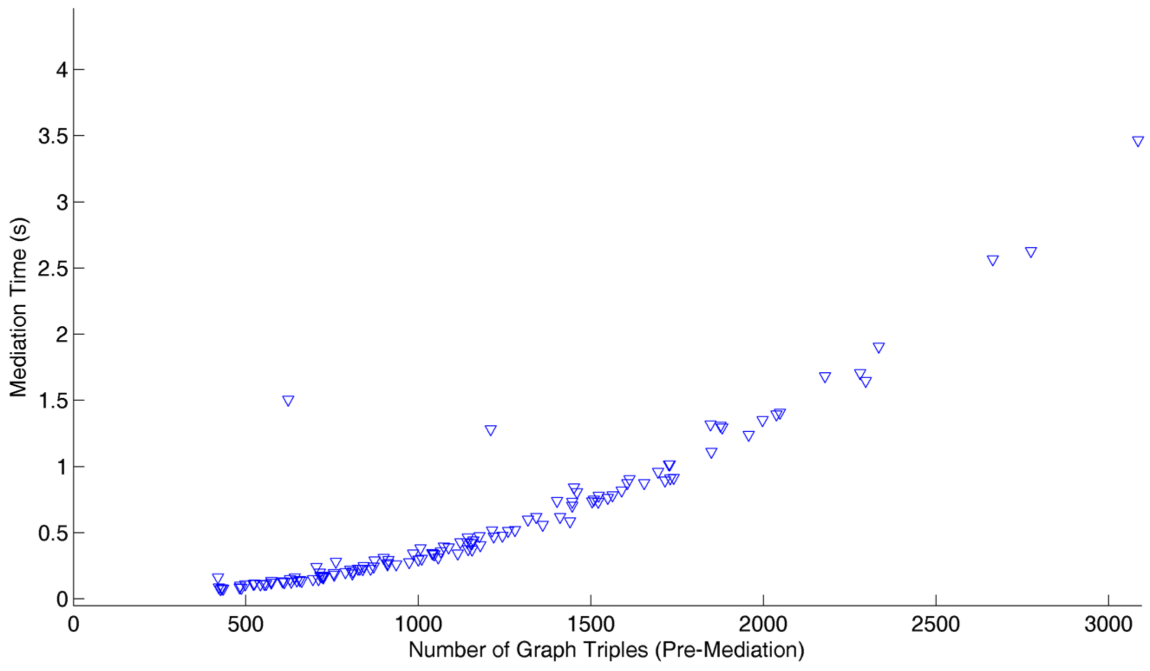


Figure 73: Measured CPU time utilized for exhaustively applying all mediation rules, plotted against the number of graph triples in the raw data graph.

inexact probabilistic reasoning process as compared to the logical reasoning process, each of which uses the same pattern formalism and pattern matching algorithms.

In addition to the CPU times, the peak *heap* memory consumption (as measured by the Java virtual machine) was stored during each experiment. Figure 74 depicts this consumption (in megabytes (MB)) as a function of the number of triples in the model to which the inexact probabilistic reasoner was exposed.

Note that memory consumption is fairly high by nature of the algorithm, which intentionally stores results of pattern matches about individual outcomes in a hashed map for processing in a final phase. In the implementation of the reasoner, MapDB is used for this purpose, which has a *hash map* implementation with a smaller overhead (and hence, smaller memory footprint per entry) and is capable of caching parts of the map on disk. Note that the Java virtual machine was limited to $7,168$MB of heap memory (7 gigabytes), hence requiring MapDB to cache any entries on the disk beyond that point. This advantage has the tradeoff of a slightly slower average

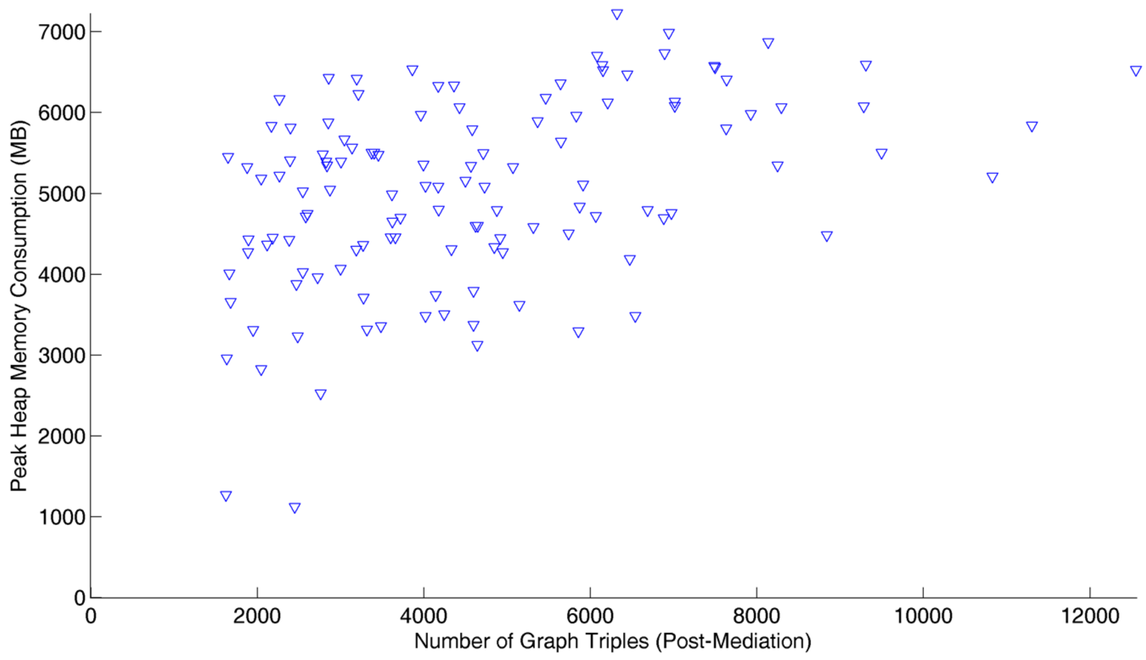


Figure 74: Peak heap memory consumption measured during the execution of the Bayesian reasoner plotted against the number of triples in the graph (with mediation rules applied exhaustively, and all RDF/RDFS inferences applied). Note that the memory consumption does not indicate a clear trend due to uncontrolled (random) garbage collection by the Java virtual machine, and due to the use of a hash table that is partially stored on the hard disk.

runtime. Finally, note that the data points are spread very far due to MapDB, the non-deterministic garbage collector, and the documented interaction between MapDB and the garbage collector (see the documentation of MapDB[7]).

8.4 Summary

In this chapter, the proposed approach to identifying inconsistencies using abductive reasoning is characterized and evaluated. Specifically, the method introduced in chapter 6 is applied to the task of identifying inconsistencies and detecting semantic overlap.

In the first part of the chapter, a quantitative analysis of the approach is performed. The chapter introduces and reviews a variety of evaluation metrics and measures from the related literature. In addition, a value-based metric for comparing different inconsistency identification approaches (and different classifiers) is introduced. As a basis for taking measurements, heterogeneous models of railway systems are automatically generated and injected with inconsistencies and imperfections (e.g., spelling mistakes and incompletenesses). Thereafter a series of experiments are conducted, and the approach is compared to a status-quo deterministic approach.

For the characterization and evaluation, inconsistency identification knowledge is created and six distinct experiments are performed. First, initial measurements are taken for subsequent comparisons. The initial measurements already indicate that the approach is capable of identifying all introduced inconsistencies. However, it also clearly indicates the suspected problem related to having to infer *intended* model overlap (see chapter 7). A follow-up experiment is conducted, where features strongly indicating the presence of an inconsistency that also have an influence on the probability of semantic overlap are removed from the reasoning knowledge. The observations made confirm the hypothesis that, under certain conditions, evidence

[7]http://www.mapdb.org

supporting inconsistency can outweigh evidence of semantic equivalence. A subsequent experiment investigates the impact of prior beliefs on the conclusions drawn by the algorithm. It was conducted by comparing the performance of a classifier with subjective beliefs, to those of a classifier with highly-informed priors (informed through previously observed instances of inconsistencies in generated models). To test the amount of information required to improve the average accuracy, reduce the number of false positives, and reduce the sensitivity of the results, a fourth experiment investigates the effects of incremental learning. It is concluded that a considerable amount of data is necessary to significantly improve the performance of the classifier. A fifth experiment investigates the impact of different cost ratios of producing false positives, true positives and false negatives. It is demonstrated that under certain circumstances, it is preferable to produce a larger number of false positives rather than invoking the cost associated with a false negatives. However, it is also shown that there is generally a trade-off that must be considered in practice. Finally, the sixth experiment explores the impact of the quantity of inconsistency identification knowledge on the performance characteristics. This is done by creating a second, much more comprehensive Bayesian network, and comparing the results from the previous experiments to those gathered using the larger network.

After conducting the experiments for purposes of characterizing the proposed methods, the inexact reasoning method is compared to a status-quo deterministic method. This is done by comparing the results gathered during the experiments to the performance of a deterministic classifier (which is implemented by a series of SPARQL queries). It is clearly demonstrated that the proposed method can lead to significantly better results.

In the last part of the chapter, the performance of the algorithmic procedures underlying the approach are evaluated. This is comprised of a theoretical complexity analysis and the presentation of empirically gathered performance results. It is shown that the theoretical results align well with the theoretical complexity. Both CPU time and memory consumption are investigated. It is concluded that, given the polynomial time behavior of the algorithm, an incremental reasoning strategy is practical.

CHAPTER IX

CONCLUSION

In this chapter, a reflection is made on the research questions and hypotheses from section 1.3 by summarizing the insights gained from the results presented in the previous chapters. The primary objective of this chapter is to evaluate the stated hypotheses by considering the evidence collected in support (and opposition) of each, and to investigate the extent to which the various research questions have been answered. In addition, the research contributions are made explicit, and limitations of the proposed methods are identified. Finally, suggestions for future work are outlined, followed by closing remarks.

9.1 Recapitulation

In chapter 1, the motivating question for this research is expressed as:

Motivating Research Question. *How, and to what extent, can inconsistencies in a collection of distributed, disparate and heterogeneous models be identified automatically?*

The primary hypothesis of the presented research is that a probabilistic approach can overcome the challenges and mitigate the limitations associated with applying state-of-the-art approaches to inconsistency identification within the context of MBSE. One such challenge is that state-of-the-art approaches do not account for the typical heterogeneity of models encountered in MBSE applications. A related challenge is the inherent and unavoidable (semantic) overlap of models, the automated detection of which is subject to very strong assumptions in current practice. Therefore, the primary focus of the presented research is the development, investigation and evaluation of an approach and its characteristics to aid in the (semi-)automated detection of semantic overlap and inconsistencies within the context of MBSE.

Because the motivating research question is too broad to be answered in a single research study, a number of more specific and focused research questions are introduced and developed in section 1.3. These research questions are re-stated in the following pages, and the degree and means to which these have been answered by the presented research is outlined. This includes presenting the accompanying hypotheses and a summary of the evidence gathered in their support (and opposition).

A fundamental basis for developing a (probabilistic) method for identifying inconsistencies in disparate, heterogeneous models is an understanding of what an inconsistency *is* and how it manifests. This insight leads to research question 1:

Research Question 1. *What are the characteristics of typical inconsistencies in engineering models? What kinds or types of inconsistencies can be identified, and what unsatisfied semantic relationships are these a result of?*

Two hypotheses are formulated as a response for this question: firstly, it is hypothesized that *"a state of inconsistency is influenced by the presence (or absence) of a number of syntactic and semantic properties that are in conflict. These syntactic and semantic properties manifest as propositions, and a configuration of conflicting propositions can be abstracted by a pattern. Furthermore, these properties can be understood to represent evidence to suggest the presence (or absence) of a particular type of inconsistency. A conflicting set of such properties (i.e., a match to a corresponding pattern) represents a manifestation of a particular type of inconsistency iff it entails the inconsistency."*. In response to the second part of research question 1, the hypothesis is formulated that *"it is possible and practical to differentiate between different types of inconsistencies. There exists both a finite, closed set, and an open, infinite set of types of inconsistencies and related types of semantic overlap"*.

The research question, and the validity of the hypotheses, is investigated primarily in chapter 4. Characteristics of inconsistencies are identified by investigating fundamentals of *consistency* within the scope of formal modeling and its application to the design of complex systems, as well as by analyzing several examples. The main supporting evidence gathered for the first hypothesis is as follows:

- The related literature reviewed in chapter 3 establishes patterns as a potentially meaningful method for representing and identifying different types of inconsistencies.
- In the first part of chapter 4, it is shown that it is generally impossible to prove consistency of a set of models, particularly within the context of MBSE. Those types of inconsistencies that *are* detectable based solely on the information and knowledge encoded in a model manifest as a part of the model. These manifestations represent identifying features which, by definition, must either be of a syntactic or semantic nature.
- Further evidence for the first hypothesis is provided in section 4.2.1. There, a number of example inconsistencies are detailed, from which identifying features of the inconsistency that are specific to the example are first extracted, and then abstracted. In all cases, the problem is shown to be reduceable to a set of conflicting assertions about semantically related entities.
- Chapter 2 and, in particular, section 2.2 detail the derivation of an inconsistency from the perspective of proof theory: if a statement and its negation can be inferred from the same formal system, the formal system is inconsistent. If an expression can be proven to not be well-formed, it is inconsistent with the formal system (given a consistent and complete formal system). It is shown that this problem is intractable, and even undecidable for most languages. This makes it non-practical within the considered context due to the semi-formal nature of most modeling languages. An alternative view is introduced in section 4.3,

where an abductive apparatus, rather than a deductive one is assumed. Abduction implies identifying the best explanation for a set of observations. Here, these observations are considered the identifying features of an inconsistency. It is demonstrated how these identifying features can be combined and abstracted, and how they represent a pattern.

- In chapter 5, it is shown that semantic and syntactic properties can be represented by propositions. Hence, types of inconsistencies manifest as configurations of conflicting propositions, further support for which is gathered throughout the application of the concepts in chapter 8.

The second part of the research question is investigated by means of reviewing classifications of inconsistencies from the related literature and by considering fundamental aspects of the definition of formal models and modeling languages. In addition, current modeling practices are considered. In summary, the main supporting evidence gathered for the second hypothesis is as follows:

- In the related literature, distinguishing between various types of inconsistencies is considered practical since it allows for better assessment of the impact of a particular discovered inconsistency, and since it aids in taking appropriate steps in resolving the inconsistency (in a later stage).

- The classifications of semantic relations and inconsistencies reviewed in section 4.2.3 are specific to languages, domains or applications. Given the observation that all inconsistencies are the result of conflicting assertions, this implies a closed set. However, similar to what most authors argue in the related literature on semantic relations, any attempt at more concretely classifying inconsistencies will necessarily lead to an open set due to the infinite nature of most languages.

Explicit knowledge of semantic relations and model overlap is crucial for identifying inconsistencies. However, specifying these manually is costly, and their automated inference remains a challenge. This insight has led to the formulation of the second research question:

Research Question 2. *How can semantic overlap and semantic relationships be identified effectively and efficiently, and to what degree can this be automated?*

The research question is primarily investigated in chapter 5, and further supported by evidence gathered in chapters 6 and 8.

Two relevant hypotheses are formulated as a response to the research question. The first hypothesis is that *"a prerequisite to an effective method for identifying inconsistencies [and semantic overlap] in heterogeneous models is the transformation of the models to a common representational formalism, thereby allowing symbolic processing across the models regardless of their nature, underlying formalisms, and organization of the encoded knowledge and information"*. In summary, the evidence gathered in support of this hypothesis is as follows:

- In chapter 5, the first hypothesis is supported by the fact that the automated identification of inconsistencies requires a symbolic processing capability across models. This does not necessarily entail translation to a common representational formalism, but such a common formalism is shown to be more effective and maintainable than ad hoc tool integrations.

- The definition of heterogeneous models given in chapter 5 also further supports the first hypothesis: heterogeneous models imply the existence of incompatible meta-models, varying formalism, different serialization formats, and different modeling tools. This increases the complexity and negatively impacts the maintainability of point-to-point integrations.

- A common representational formalism has shown promise in chapters 5 and 6, where the patterns defined for identifying inconsistencies span information and knowledge in multiple models. This requires a pattern formalism that is capable of incorporating data from models irrespective of their nature, formalism and organization of encoded knowledge and information.

- The value and practicality of using a common formalism is also supported by the application and evaluation in chapter 8, where the methods proposed in chapters 5 and 6 are successfully applied.

The second hypothesis is that *"an effective method for identifying inconsistencies [and semantic overlap] throughout the life-cycle that is capable of drawing conclusions from an incomplete (but continuously refined) description of a system should be based on Bayesian updating."* This leads to the development of the method detailed in chapter 6, and an investigation of the results from applying this method in chapter 8. The main evidence gathered in support of the hypothesis is as follows:

- How Bayesian updating can be applied within the context of reasoning about syntactic and semantic properties of models is detailed in chapter 6. The developed concept acts as supporting evidence since it demonstrates the feasibility of such a method. The technical feasibility is demonstrated through the development of a proof-of-concept tool support in chapter 8.

- The developed concepts are applied to an example inconsistency identification scenario in chapter 8 (specifically section 8.2.5). The results clearly indicate the capability of the method to identify inconsistencies and semantic overlap.

- In the same section (8.2.5), it is also demonstrated that the approach is capable of overcoming the challenge of detecting an *intended* semantic overlap of inconsistent entities. This challenge is first identified in chapter 7.

- The experiment conducted in section 8.2.6 clearly illustrates that a Bayesian-updating-based method for identifying inconsistencies in sets of potentially incomplete and inconsistent models is capable of identifying more inconsistencies than a comparable deterministic approach. Particularly, the value-based comparison indicates that the value of applying probabilistic reasoning can be considerably higher than the value of deterministic reasoning.
- Discussions and references to the literature in chapter 7, and insights gained from the experiments conducted in chapter 8 indicate improved maintainability of the reasoning knowledge over using a set of deterministic rules.

Finally, the third research question is formulated as follows:

Research Question 3. *What is an effective way of aiding modelers in the process of efficiently detecting inconsistencies in a set of collaboratively developed, heterogeneous and distributed formal engineering models? How can we improve upon the status quo of rule-based approaches?*

Given that the approach to both the identification of semantic overlap and inconsistencies is the same, the third research question is closely related to the second. However, it focuses more specifically on the identification of *inconsistencies*, and the interpretation and continuously learning from the results of applying the proposed inference method. Therefore, for the first hypothesis – that Bayesian updating is an effective method for the purpose of identifying inconsistencies – evidence has already been presented. The second hypothesis is that *"an effective approach to inconsistency identification should consider the aspect of learning from experience. Granting hypothesis 4, methods for encoding, integrating and processing relevant past experience and expert knowledge for the purpose of refining inconsistency identification knowledge should make use of (Bayesian) machine learning"*. The following evidence is collected in support of this hypothesis:

- In chapter 8 (specifically section 8.2.5.5) it is demonstrated that using Bayesian learning (i.e., generating data cases and automatically updating Bayesian network parameters) has a positive impact on several metrics, most notably the number of FPs produced.

- Identifying a suitable *cutoff probability* is identified in chapter 7 as an appropriate classification measure to aid a human in interpreting the results. Not using such a heuristic to reduce the number of inferred inconsistencies can lead to a large number of false positives. This is demonstrated in chapter 8. Different strategies for presenting the results (e.g., clustering and rank ordering) to aid in interpreting inference results are discussed in chapter 7.

- The research question is addressed, in part, by the introduction of a semantic abstraction mechanism in chapter 5. This semantic abstraction mechanism provides a basic interface between models by introducing (and translating to) terms whose semantics are more abstract than the terms used in (a subset of) the models being analyzed. Thereby, the formulation of patterns (an essential part of inconsistency identification knowledge) is also made more manageable. This is given through the introduction of terms with varying levels of semantic abstraction, thereby enabling higher level reasoning over a set of heterogeneous models.

- From the perspective of efficiency, the need for an incremental reasoning strategy is identified, and a corresponding algorithm to address this is developed in chapter 6. Empirical performance measurements presented in chapter 8 clearly show the gain in efficiency by this strategy.

In opposition of the second hypothesis, it is shown in chapter 8 that, in order to *significantly* improve inference results, a considerable number of data cases from which to learn must be available. This is a general difficulty of applying Bayesian learning in practice.

9.2 Contributions

Three primary contributions are made in this dissertation: a common representational formalism for heterogeneous models (chapter 5), a probabilistic inexact (abductive) reasoning method over models represented in this common formalism (chapter 6), and the application of this Bayesian inference based method to the problem of identifying probable inconsistencies and semantic overlap. In addition, a number of secondary contributions have been made. In the following sections, important aspects of each of these contributions is briefly discussed.

9.2.1 Common Representational Formalism for Heterogeneous Models

In MBSE, the use of heterogeneous models is omnipresent. This model heterogeneity manifests itself in three dimensions: firstly, the different types of models and their nature (specification or analysis, process or artifact); secondly, the incompatibility of meta-models; thirdly, an extensive tool landscape with very limited integration, which hinders their integration. However, reasoning about inconsistencies in such heterogeneous models requires the ability to perform symbolic processing across model boundaries. For this purpose, a conceptual basis for a common representational formalism is developed.

The common representational formalism is introduced in chapter 5. Aspects of both representing syntactic and semantic structures are discussed. The focus is on developing a basis for capturing the information and knowledge contained in models in a propositional form. Directed, labeled multi-graphs are identified as a suitable mathematical structure for this purpose, where (atomic) propositions are represented by graph triples. Graph queries and graph transformation rules are presented as mechanisms for information retrieval and manipulation.

A mere translation of formal models to this common representational formalism alone is not sufficient for purposes of integration. To enhance reasoning capabilities, the concept of *mediation* is introduced. Mediation is a necessary basis for heterogeneous models to interface and interact. A key concept of the approach is the exploitation of language- and domain-specific concepts to infer semantic information for higher-level reasoning applications.

9.2.2 Method for Probabilistic Inexact (Abductive) Reasoning over Graph-Based Models

The second major contribution of this dissertation is a method for probabilistic inexact reasoning over graph-based models. The method can be classified as an *abductive* reasoning approach, since it is *explanatory* in nature. As explained in chapters 4 and 6, abductive reasoning has several advantages over *deductive* reasoning within the context of MBSE. For instance, while the conclusions reached by an abductive reasoning apparatus are not always logically correct, the decidability of the abductive approach guarantees that an answer can be given, and in finite time. In addition, a provably complete and consistent definition of an underlying formal system is not required.

Bayesian probability theory along with the concepts developed as part of the common representation formalism, form the basis for the method, which is developed in detail in chapter 6. A fundamental idea is the association of graph patterns with target space values of random variables. Matches to the relevant patterns indicate set membership in the pre-images of the respective random variables. Chapter 7 discusses the necessity of these matches having to lead to mutually exclusive result sets. Thereby, evidence can be collected for updating a prior belief about the existence of a semantic property (such as inconsistency or semantic equivalence) by means of pattern matching. Updating this belief is done by constructing and performing probabilistic inference in a Bayesian network. In chapter 8 it is shown that the proposed approach can, within the context of MBSE, lead to significantly better results than using deterministic reasoning mechanisms.

9.2.3 Automated Identification of Probable Inconsistencies and Semantic Overlap in Graph-Based Models using Abductive Reasoning

A third major contribution of this dissertation is the application of Bayesian inference to inconsistency identification and semantic overlap detection within the context of MBSE. Specifically, this includes applying the developed probabilistic reasoning approach to the problem of automatically identifying inconsistencies and, in the process, to the automated inference of a probable semantic overlap.

Chapter 7 discusses important characteristics, properties and considerations of applying inexact reasoning for the task of identifying inconsistencies. This includes a summary of the knowledge that should be acquired as part of identifying a particular type of inconsistency. The impact of the characteristics of the approach are investigated in more detail in chapter 8, where the approach is applied to a concrete case study. As part of the evaluation, the approach is also compared to a deterministic approach in section 8.2.6.

9.2.4 Secondary Contributions

Several secondary contributions were made as part of this dissertation. These are briefly summarized in the following pages.

Characterization & Classification of Types of Inconsistencies in Heterogeneous Models In the related literature, numerous definitions for the term *"inconsistency"*, and a variety of classifications of types of inconsistencies can be identified. However, most of these definitions and classifications stem from software engineering research, and are closely aligned with concepts from UML. However, little to no work has been done towards characterizing and classifying inconsistencies within the context of Model-Based Systems Engineering. Since this characterization is a vital and fundamental basis for developing a method for identifying inconsistencies within the context of MBSE, an investigation into the fundamental characteristics of inconsistencies as well as a definition for the term "inconsistency" within the scope of this work is presented in chapter 4. This is followed by a classification of inconsistencies in section 4.2.3. The important insight gained is that it is practical to differentiate between different types of inconsistencies, and that it is beneficial to reason about inconsistencies in an abductive, rather than a deductive fashion.

Semantic Abstraction Mechanism For the purpose of enabling higher level reasoning, a semantic abstraction mechanism is presented as part of the developed fundamentals of the reasoning framework. The mechanism is presented in more detail in chapter 5. A key idea of the semantic abstraction mechanism is the introduction of, and mediation (translation) between a number of language- and domain-specific vocabularies, as well as a common base vocabulary. While binding concepts from different languages together at a higher level of abstraction, and enabling a basic interface between models based on abstract types, the abstraction mechanism also allows for definitions of patterns that refer to concepts that span different modeling languages.

Metrics and Procedures for Evaluating Probabilistic Inconsistency & Semantic Overlap Classifiers The method for inexact reasoning about inconsistencies and semantic overlap developed in chapters 6 and 7 is applied to a case study in chapter 8. There, the overall approach is also characterized and evaluated. The procedures used are a secondary contribution, since they are applicable to evaluating any set of inconsistency identification knowledge. This also includes the identification of appropriate measures and metrics. A number of measures from the related literature are reviewed (e.g., recall, precision and F-measure), all of which are commonly applied within the context of machine learning for evaluating similar approaches.

While informative for making assertions about certain characteristics of applying the approach, none of the measures were found to be suitable for comparative purposes and for determining the true *value* of the approach. One reason for other measures not being suitable is their inability to demonstrate the potentially large impact of false negatives (i.e., inconsistencies or semantic equivalences that remain undiscovered). This led to the definition of a *value-based* metric, which takes into account the cost incurred by the production of true and false positives, as well as the strongly adverse impact of false negatives. Using this value-based metric, different sets of inconsistency identification knowledge are compared. In addition, a comparison to a state-of-the-art deterministic approach is performed. As a basis for measurements, sets of heterogeneous models are generated.

Java Library for Constructing, Performing Inference in, and Learning Bayesian Network Parameters As part of the research, a Java library for representing, performing inference in, and learning the parameters of Bayesian networks was developed. This library can be claimed as a secondary contribution, since, unlike other notable Java-based Bayesian network libraries such as `Weka` [5], `Mahout` [167] and `JavaBayes`[1], it allows for *Dirichlet distributions* to be imposed over network parameters (rather than just specifying values in conditional probability tables) and supports learning of *general Bayesian networks* (rather than just naïve Bayes model learning). At the time of writing this dissertation, the library has not been released as open source, but plans exist to do so in the near future.

[1]http://www.cs.cmu.edu/ javabayes/

9.3 Limitations & Future Work

As evident from the results and insights gained during the quantitative evaluation, the proposed approach has value within the context of MBSE under certain conditions. However, a number of limitations have also been identified in the process. Overcoming these requires further research, which is considered outside the scope of this dissertation. To guide further exploration of the developed concepts, notable limitations of the proposed approach and potential areas for future work are presented in this section.

9.3.1 Acquiring & Maintaining Inconsistency Identification Knowledge

As discussed in chapter 7, the acquisition and elicitation of the inconsistency identification knowledge is expected to be a costly process. This can significantly reduce the value of applying the approach. Relative to other state-of-the-art inconsistency identification processes, more knowledge must be acquired. For instance, degrees of belief on the occurrence of a relatively large number of events must be elicited. Furthermore, if not already available, mediation vocabularies and accompanying inference rules must be defined. Non-negligible is also the process of having to elicit and verify a series of graph patterns: for each, it must be shown that it (a) implies a particular intended semantic or syntactic property with sufficient confidence and (b) that matches to the pattern can be guaranteed to be mutually exclusive of other patterns associated with the same random variable. How this can be done in an efficient manner is not explored in great depth, but should be investigated in future work.

Non-negligible are also costs associated with refining inconsistency identification knowledge. This is primarily due to the number of false positives being the only discernible indicator about the *goodness* of the current state of inconsistency identification knowledge. By their nature, false negatives cannot be detected without a thorough analysis of the underlying model data. Future work should investigate this issue of determining at what point a refinement of inconsistency identification knowledge is valuable.

Within the scope of the dissertation, it is assumed that inconsistency identification knowledge is created by humans. A number of methods from the literature are proposed in chapter 7 that have the potential of aiding in the elicitation of such knowledge. Also discussed is the potential for reuse of the acquired knowledge across different application scenarios (see section 7.2.5). However, what has not been explored, but would serve as an interesting basis for further research, is the automated extraction of patterns from hand-labeled sets of inconsistent models using techniques from data mining. Given the availability of such sets, and given that they are accurate reflections of the models to which the inconsistency identification knowledge will be applied, this could reduce the cost associated with the acquisition of reasoning knowledge.

9.3.2 Scalability & Performance

In chapter 8 (and in particular section 8.3.2), empirical performance measurements are presented. These are compared to the theoretical complexity derived in section 8.3.1. Recall that the derivation of the theoretical complexity made a number of strong assumptions, and the result is only valid within the scope of the assumptions that (a) the underlying graph model is a directed graph and (b) only discrete random variables are utilized (inference in Bayesian networks with continuous random variables is computationally more complex). Given the polynomial-time behavior of the algorithm (polynomial with respect to the number of graph triples), an incremental strategy such as the one proposed in section 6.3.1 is inevitable. However, for very large graphs, and for more than just one type of inconsistency (which would involve a series of Bayesian networks), the runtime could become a severe limitation.

Future work should include further development of the underlying algorithms for improving performance. For instance, one possible improvement is the exploitation of the independence assumptions in the Bayesian network to more intelligently sequence the procedure of matching patterns, thereby avoiding unnecessary computation cycles and graph searches. Concretely, this avoids attempting to seek out evidence that, when used for updating a particular belief, has no influence on the result (given the other already available evidence). In addition, under certain conditions pattern matching operations in graphs can be parallelized (see, e.g., the work by Taentzer [211]). Note that these measures do not reduce the theoretical complexity, but can significantly reduce the actual inference time.

In addition to the computational complexity, the proposed incremental algorithms are also highly memory-intensive (see the empirical measurements in section 8.3.2). This is under the premise that for larger scale applications, technologies such as distributed computing can be used to increase the available storage (whether in-memory or on disk). For instance, the Hadoop [196] infrastructure and accompanying distributed in-memory file system could serve as a promising basis for this[2].

Ultimately, from a computational standpoint, scalability and performance are limited by the available computational resources (primarily CPU frequency), and the optimality of the underlying algorithms. Beyond just computational considerations, the cost of gathering the initial reasoning knowledge is non-negligible (see previous section). Whether applying the proposed approach is valuable strongly depends on the context in which it is applied. This context includes the expected impact of false negatives, the degree of inconsistency of the models being reasoned over and the confidence in the reasoning knowledge being applied.

[2]Hadoop's file system (HDFS) is similar in nature to the proprietary distributed file systems used by popular search engines. By keeping the file system in-memory, these enable quick access to very large amounts of data, typically spread over a number of physical nodes (queries over this data are distributed).

9.3.3 Characterization & Evaluation (Validation) under Controlled Conditions

A limitation of the results presented in this dissertation – particularly those gathered when evaluating the approach – is the fact that they have been produced under controlled (laboratory) conditions. In chapter 8, the basis for the experiments are randomly generated sets of models that are algorithmically injected with (one or more types of) inconsistencies, incompletenesses and (what are deemed common) features representing human imperfections (such as misspellings and the use of synonyms). Two major limitations of this procedure are the potential impact on the conclusions by the introduction of a possible bias, and the relatively limited scope of the case study.

Given that the same person has created both the algorithmic generation of the data being reasoned over and the reasoning knowledge, a bias is unavoidably introduced. An effort to minimize the degree to which this bias influences the conclusions is the introduction of a very large number of possible combinations of any one of several types of inconsistencies and imperfections that are randomly introduced. That this has succeeded at least to some degree can be observed from the (imperfect) results gathered, where the measurements gathered using a Bayesian network with the *true* values for the Bayesian network parameters are compared to one utilizing subjective network parameters.

A second limitation is the scope of the models generated. While three types of models are involved, and the true number of models is random, their expressiveness is comparatively small. Future work should include the application of the proposed methods to a larger case study – ideally within an industrial context. To fully assess the value in realistic scenarios, state-of-the-art approaches (such as the deterministic pattern based approach) should be investigated in parallel, and within a system development context of similar scope. Care needs to be exerted in avoiding an introduction of a bias, and in diligently recording the costs involved for both approaches. This allows a better assessment of the value of the proposed approach outside the controlled conditions.

9.3.4 Reasoning Scope

The approach, as presented in chapter 6, assumes reasoning within a highly localized context. That is, for a particular outcome (say, a pair of properties), any probable semantic properties (say, the probability of the properties being inconsistent) depends solely on observations that can be made within the local graph context. That is, results of previously made inferences are not accessible to subsequent ones. To exemplify this, assume the existence of two entities A and B, each owning a number of properties $p_{A,i}$ and p_{B_j}. Now assume that the probability of a particular pair of properties of A and B being semantically equivalent should depend, among other evidence, on the the probability of the other properties being semantically equivalent. Accounting for such knowledge requires access to previously made inferences and is not currently supported. Future work should investigate the value of performing reasoning on such a *global* scale.

9.3.5 Application of Concepts Beyond Inconsistency Identification

Given that the approach to probabilistic inexact reasoning (introduced in chapter 6) is not limited to the inference of inconsistencies, future work should include the application of the proposed concepts outside the realm of inconsistency identification. Generally, the approach can be applied in any scenario where the data being reasoned over is representable by a graph.

9.4 Closing Remarks

In this dissertation, a probabilistic approach to identifying inconsistencies is developed and, in the process, so is the automated inference of semantic overlap using abductive reasoning in sets of heterogeneous models. The approach represents a novel view on a problem that, in the related literature, is primarily considered from a deterministic perspective and using deductive or inductive mechanisms.

Even though a number of limitations are identified in section 9.3, the contributions from this research are nonetheless significant within the context of identifying inconsistencies and semantic overlap with computational support, and Model-Based Systems Engineering. Based on the controlled conditions under which results were obtained and conclusions drawn, one cannot claim that the investigated approach represents the best (or rather: most valuable) solution for identifying inconsistencies in heterogeneous models. However, the experiments performed, results gathered and insights gained show that the method represents a significant improvement over the status quo. The gathered results are deemed useful for future investigations extending the presented method and developing other automated approaches to inconsistency identification.

APPENDIX A

BAYESIAN NETWORKS, CONDITIONAL PROBABILITY DISTRIBUTIONS & DETERMINISTIC CLASSIFIERS USED IN EVALUATION OF APPROACH

A.1 Compact Bayesian Network

In the following, the structure and elicited beliefs on the network parameters of the *compact Bayesian network* first introduced in section 8.2 are presented in detail. This network was used as part of the quantitative evaluation of the approach.

A.1.1 Structure

Table 9 lists the random variables, associated target space values, and patterns for this network. For a visualization of the network, and the influence relations between the random variables, refer to figure 46.

The network is used for reasoning about the inconsistency and semantic equivalence of distinct pairs of properties (p_i, p_j) where $p_i \neq p_j$. The base pattern is (?p1 rdf:type bso:Property) (?p2 rdf:type bso:Property) notEqual(?p1, ?p2) (i.e., for a successful match to the pattern, the nodes bound to node variables ?p1 and ?p2 must be non-equal (have different URIs), and both bound nodes must be of type base:Property). Note that the functor *similarString* calculates the Levenshtein-distance-based similarity score introduced in section 8.2.5.1 and returns `true` if it is within the specified range (lower value is exclusive, except for the case of 0) (the argument *'i'* indicates case-insensitivity). The functors *synonym* and *notSynonym* return `true` if the words are synonyms or syntactically equal. All other functors are built-in to Apache Jena.

Table 9: Random variables, target space values and associated patterns of the compact Bayesian network used in evaluation.

Random Variable	**Target Space Value**	**Pattern**
AreSimilarParentEntities	*Entities X and Y which are parents of Properties P1 and P2 are equivalent*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x sem:equivalentTo ?y)
	Entities X and Y which are parents of Properties P1 and P2 are not equivalent	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x sem:differentFrom ?y)
AreEquivalentProperties	*Properties P1 and P2 are equivalent*	(?p1 sem:equivalentTo ?p2)
	Properties P1 and P2 are not equivalent	(?p1 sem:differentFrom ?p2)
AreInconsistentProperties	*Properties P1 and P2 are inconsistent*	(?p1 incon:inconsistent ?p2)
	Properties P1 and P2 are not inconsistent	(?p1 incon:notInconsistent ?p2)
ConstraintSimilarity	*Properties P1 and P2 have equivalent constraints*	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:unitType ?ut1) (?c2 base:unitType ?ut2) (?c1 base:value ?v1) (?c2 base:value ?v2) equal(?ut1, ?ut2) equal(?v1, ?v2)

Continued on next page

Table 9 (continued)

Random Variable	**Target Space Values**	**Patterns**
	Properties P1 and P2 have non-unit constraints with equal values	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) noValue(?c1 base:unitType) (?c1 base:value ?v1) (?c2 base:value ?v2) equal(?v1, ?v2)
	Properties P1 and P2 have non-unit constraints with non-equal values	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) noValue(?c1 base:unitType) (?c1 base:value ?v1) (?c2 base:value ?v2) notEqual(?v1, ?v2)
	Properties P1 and P2 have no constraints with equal units and values	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:unitType ?ut1) (?c1 base:value ?v1) noValue(?c2 base:unitType ?ut1) noValue(?c2 base:value ?v1)

Continued on next page

Table 9 (continued)

Random Variable	**Target Space Values**	**Patterns**
EntityNamesSimilarityScore	*The names of X and Y have a similarity score of 0.8 to 1.0*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, ’0.8’, ’1.0’, ’i’)
	The names of X and Y have a similarity score of 0.6 to 0.8	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, ’0.6’, ’0.8’, ’i’)
	The names of X and Y have a similarity score of 0.4 to 0.6	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, ’0.4’, ’0.6’, ’i’)

Continued on next page

Table 9 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of X and Y have a similarity score of 0.2 to 0.4	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, ’0.2’, ’0.4’, ’i’)
	The names of X and Y have a similarity score of 0.0 to 0.2	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, ’0.0’, ’0.2’, ’i’)
EntityTypeRelationNamesSimilarityScore	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, ’0.8’, ’1.0’, ’i’)

Continued on next page

Table 9 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, ’0.6’, ’0.8’, ’i’)
	The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, ’0.4’, ’0.6’, ’i’)
	The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, ’0.2’, ’0.4’, ’i’)
	The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, ’0.0’, ’0.2’, ’i’)

A.1.2 Network Parameters

Table 10 documents the beliefs on the Bayesian network parameters, elicited as *Dirichlet* distributions (see section 2.3). Beliefs were elicited in the form of a willingness to bet on an event, and in the manner described in section 7.2.4. The following is a sample elicitation question used in the process:

> *"Say the outcome of an experiment is a randomly selected pair of properties* $(P1, P2)$*, for which it is known that their names are similar (with a similarity score value between 0.8 and 1.0), their owning entities* X *and* Y *have similar names (with a similarity score value between 0.8 and 1.0), none of their constraints have unit types specified, and none of these have equal values. How much are you prepared to stake (in fractions of \$1) in a gamble where you win \$1 if the properties* $P1$ *and* $P2$ *are inconsistent, and lose your stake if they are not (i.e., win \$0)?"*

To capture the belief as a probability distribution, *Dirichlet* distributions are used. *Beta* and *Dirichlet* distributions are particularly well suited for eliciting beliefs due to the natural way of specifying the parameters [152]. Note that a two-parameter Dirichlet distribution is equivalent to a Beta distribution.

Table 10: Elicited beliefs on the network parameters of the compact Bayesian network used for evaluating the approach.

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*Properties P1 and P2 have equivalent constraints*, *The names of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.98, 0.02)
AreSimilarParentEntities	*Properties P1 and P2 have equivalent constraints*, *The names of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.8, 0.2)
AreSimilarParentEntities	*Properties P1 and P2 have equivalent constraints*, *The names of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.6, 0.4)
AreSimilarParentEntities	*Properties P1 and P2 have equivalent constraints*, *The names of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.4, 0.6)
AreSimilarParentEntities	*Properties P1 and P2 have equivalent constraints*, *The names of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.2, 0.8)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with equal values*, *The names of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.8, 0.2)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with equal values*, *The names of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.7, 0.3)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with equal values, The names of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.5, 0.5)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with equal values, The names of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.3, 0.7)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with equal values, The names of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.15, 0.85)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with non-equal values, The names of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.65, 0.35)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with non-equal values, The names of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.45, 0.55)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with non-equal values, The names of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.35, 0.65)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with non-equal values, The names of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.15, 0.85)
AreSimilarParentEntities	*Properties P1 and P2 have non-unit constraints with non-equal values, The names of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.05, 0.95)
AreSimilarParentEntities	*Properties P1 and P2 have no constraints with equal units and values, The names of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.6, 0.4)
AreSimilarParentEntities	*Properties P1 and P2 have no constraints with equal units and values, The names of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.4, 0.6)
AreSimilarParentEntities	*Properties P1 and P2 have no constraints with equal units and values, The names of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.3, 0.7)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*Properties P1 and P2 have no constraints with equal units and values*, *The names of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.1, 0.9)
AreSimilarParentEntities	*Properties P1 and P2 have no constraints with equal units and values*, *The names of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.01, 0.99)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0*, *Properties P1 and P2 have equivalent constraints*, *Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.99999, 1.0E-5)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0*, *Properties P1 and P2 have equivalent constraints*, *Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.3, 0.7)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.99, 0.01)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.25, 0.75)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.85, 0.15)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.05, 0.95)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.82, 0.18)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.01, 0.99)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8, Properties P1 and P2 have equivalent constraints, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.99, 0.01)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8, Properties P1 and P2 have equivalent constraints, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.28, 0.72)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.95, 0.05)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.2, 0.8)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.82, 0.18)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.03, 0.97)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8*, *Properties P1 and P2 have no constraints with equal units and values*, *Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.79, 0.21)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8*, *Properties P1 and P2 have no constraints with equal units and values*, *Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.008, 0.992)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6*, *Properties P1 and P2 have equivalent constraints*, *Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.8, 0.2)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6*, *Properties P1 and P2 have equivalent constraints*, *Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.2, 0.8)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.75, 0.25)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.1, 0.9)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.5, 0.5)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.01, 0.99)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.4, 0.6)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.003, 0.997)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4, Properties P1 and P2 have equivalent constraints, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.6, 0.4)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4, Properties P1 and P2 have equivalent constraints, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.05, 0.95)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4*, *Properties P1 and P2 have non-unit constraints with equal values*, *Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.7, 0.3)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4*, *Properties P1 and P2 have non-unit constraints with equal values*, *Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.1, 0.9)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4*, *Properties P1 and P2 have non-unit constraints with non-equal values*, *Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.3, 0.7)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4*, *Properties P1 and P2 have non-unit constraints with non-equal values*, *Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.01, 0.99)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.1, 0.9)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.001, 0.999)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have equivalent constraints, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.5, 0.5)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have equivalent constraints, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.03, 0.97)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.6, 0.4)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have non-unit constraints with equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.02, 0.98)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.1, 0.9)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have non-unit constraints with non-equal values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(0.005, 0.995)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are equivalent*	Dir(0.05, 0.95)
AreEquivalentProperties	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2, Properties P1 and P2 have no constraints with equal units and values, Entities X and Y which are parents of Properties P1 and P2 are not equivalent*	Dir(1.0E-4, 0.9999)
AreInconsistentProperties	*Properties P1 and P2 have equivalent constraints, Properties P1 and P2 are equivalent*	Dir(1.0E-6, 0.999999)
AreInconsistentProperties	*Properties P1 and P2 have equivalent constraints, Properties P1 and P2 are not equivalent*	Dir(1.0E-6, 0.999999)
AreInconsistentProperties	*Properties P1 and P2 have non-unit constraints with equal values, Properties P1 and P2 are equivalent*	Dir(0.05, 0.95)
AreInconsistentProperties	*Properties P1 and P2 have non-unit constraints with equal values, Properties P1 and P2 are not equivalent*	Dir(1.0E-6, 0.999999)

Continued on next page

Table 10 (continued)

Random Variable	**Parent Values**	**Distribution**
AreInconsistentProperties	*Properties P1 and P2 have non-unit constraints with non-equal values*, *Properties P1 and P2 are equivalent*	Dir(0.8, 0.2)
AreInconsistentProperties	*Properties P1 and P2 have non-unit constraints with non-equal values*, *Properties P1 and P2 are not equivalent*	Dir(1.0E-6, 0.999999)
AreInconsistentProperties	*Properties P1 and P2 have no constraints with equal units and values*, *Properties P1 and P2 are equivalent*	Dir(0.99999, 1.0E-5)
AreInconsistentProperties	*Properties P1 and P2 have no constraints with equal units and values*, *Properties P1 and P2 are not equivalent*	Dir(1.0E-6, 0.999999)
ConstraintSimilarity	None	Dir(5.0E-4, 0.001, 0.9, 0.0985)
EntityNamesSimilarityScore	None	Dir(1.0E-4, 0.005, 0.1, 0.3, 0.5949)
EntityTypeRelationNamesSimilarityScore	None	Dir(0.05, 0.1, 0.25, 0.4, 0.2)

A.2 Comprehensive Bayesian Network

In the following, the structure and elicited beliefs on the network parameters of the *comprehensive Bayesian network* first introduced in section 8.2 is detailed. This network was used as part of the quantitative evaluation of the approach.

A.2.1 Structure

Table 11 lists the random variables, associated target space values, and patterns for this network. For a visualization of the network, and the influence relations between the random variables, refer to figure 62.

The network is used for reasoning about the inconsistency and semantic equivalence of distinct pairs of properties (p_i, p_j) where $p_i \neq p_j$. The base pattern is (?p1 rdf:type bso:Property) (?p2 rdf:type bso:Property) notEqual(?p1, ?p2) (i.e., for a successful match to the pattern, the nodes bound to node variables ?p1 and ?p2 must be non-equal (have different URIs), and both bound nodes must be of type base:Property). Note that the functor *similarString* calculates the Levenshtein-distance based similarity score introduced in section 8.2.5.1 and returns `true` if it is within the specified range (lower value is exclusive, except for the case of 0) (the argument *'i'* indicates case-insensitivity). The functors *synonym* and *notSynonym* return `true` if the words are synonyms or syntactically equal. All other functors are built in to Apache Jena.

Table 11: Random variables, target space values and associated patterns of the comprehensive Bayesian network used in evaluation.

Random Variable	**Target Space Value**	**Pattern**
AreSimilarParentEntities	*Entities X and Y which are parents of Properties P1 and P2 are equivalent*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x sem:equivalentTo ?y)
	Entities X and Y which are parents of Properties P1 and P2 are not equivalent	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x sem:differentFrom ?y)
AreEquivalentProperties	*Properties P1 and P2 are equivalent*	(?p1 sem:equivalentTo ?p2)
	Properties P1 and P2 are not equivalent	(?p1 sem:differentFrom ?p2)
AreInconsistentProperties	*Properties P1 and P2 are inconsistent*	(?p1 incon:inconsistent ?p2)
	Properties P1 and P2 are not inconsistent	(?p1 incon:notInconsistent ?p2)
EntityNamesSimilar	*Entities X and Y have similar names*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x sem:similarName ?y)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	Entities X and Y have dissimilar names	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x sem:dissimilarName ?y)
EntityTypesSimilar	*Entities X and Y have similar types*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?yt rdf:type base:Element) (?xt sem:equivalentTo ?yt)
	Entities X and Y have dissimilar types	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?yt rdf:type base:Element) (?xt sem:differentFrom ?yt)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
EntityTypeNamesSimilar	*Entities X and Y have types with similar names*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?xt base:name ?ytn) (?xt sem:similarName ?yt)
	Entities X and Y have types with dissimilar names	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?xt base:name ?ytn) (?xt sem:dissimilarName ?yt)
RelationKindsAreSimilar	*The property kinds of P1 and P2 are similar*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 sem:equivalentTo ?pt2)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The property kinds of P1 and P2 are dissimilar	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 sem:differentFrom ?pt2)
RelationTypeNamesSimilar	*The types of properties P1 and P2 have similar names*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) (?pt1n sem:similarName ?pt2n)
	The types of properties P1 and P2 have dissimilar names	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) (?pt1n sem:dissimilarName ?pt2n)
HaveSimilarDomain	*The domains of P1 and P2 are similar*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pt1d) (?pt2 base:domain ?pt2d) (?pt1d sem:equivalentTo ?pt2d)
	The domains of P1 and P2 are dissimilar	(?p1 base:type ?pt1) (?p2 base:type ?pt2)(?pt1 base:domain ?pt1d) (?pt2 base:domain ?pt2d) (?pt1d sem:differentFrom ?pt2d)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
HaveSimilarRange	*The ranges of P1 and P2 are similar*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r sem:equivalentTo ?pt2r)
	The ranges of P1 and P2 are dissimilar	(?p1 base:type ?pt1) (?p2 base:type ?pt2)(?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r sem:differentFrom ?pt2r)
SimilarConnectedInstances	*The connected instances through P1 and P2 are similar*	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v) (?c2 base:value ?v)
	The connected instances through P1 and P2 are not similar	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) isLiteral(?v1) notEqual(?v1, ?v2)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
EntityNamesSynonymsOrEqual	*Entities X and Y have names that are known synonyms or are equal*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) synonyms(?n1, ?n2)
	Entities X and Y have names that are not known synonyms and are not equal	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) notSynonyms(?n1, ?n2)
EntityNamesSimilarityScore	*The names of X and Y have a similarity score of 0.8 to 1.0*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, '0.8', '1.0', 'i')

Continued on next page

Table 11 (continued)

Random Variable	Target Space Values	Patterns
	The names of X and Y have a similarity score of 0.6 to 0.8	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, '0.6', '0.8', 'i')
	The names of X and Y have a similarity score of 0.4 to 0.6	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, '0.4', '0.6', 'i')
	The names of X and Y have a similarity score of 0.2 to 0.4	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, '0.2', '0.4', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of X and Y have a similarity score of 0.0 to 0.2	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type base:Element) (?y rdf:type base:Element) (?x base:name ?n1) (?y base:name ?n2) similarString(?n1, ?n2, '0.0', '0.2', 'i')
EntityTypeNamesSynonymousOrEqual	*Entities X and Y have types with synonymous or equal names*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) synonyms(?xtn, ?ytn)

Continued on next page

Table 11 (continued)

Random Variable	Target Space Values	Patterns
	Entities X and Y have types with non synonymous and non equal names	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) notSynonyms(?xtn, ?ytn)
EntityTypeNamesSimilarityScore	*The names of the types of X and Y have a similarity score of 0.8 to 1.0*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) similarString(?xtn, ?ytn, ’0.8’, ’1.0’, ’i’)

Continued on next page

Table 11 (continued)

Random Variable	Target Space Values	Patterns
	The names of the types of X and Y have a similarity score of 0.6 to 0.8	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) similarString(?xtn, ?ytn, ’0.6’, ’0.8’, ’i’)
	The names of the types of X and Y have a similarity score of 0.4 to 0.6	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) similarString(?xtn, ?ytn, ’0.4’, ’0.6’, ’i’)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the types of X and Y have a similarity score of 0.2 to 0.4	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) similarString(?xtn, ?ytn, '0.2', '0.4', 'i')
	The names of the types of X and Y have a similarity score of 0.0 to 0.2	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?xt base:name ?xtn) (?yt rdf:type base:Element) (?yt base:name ?ytn) similarString(?xtn, ?ytn, '0.0', '0.2', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
EntityTypesSame	*Entities X and Y have the same types*	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?yt rdf:type base:Element) equal(?xt, ?yt)
	Entities X and Y do not have the same types	(?x base:contains ?p1) (?y base:contains ?p2) (?x rdf:type ?xt) (?y rdf:type ?yt) notEqual(?xt, base:Element) notEqual(?yt, base:Element) (?xt rdf:type base:Element) (?yt rdf:type base:Element) notEqual(?xt, ?yt)
ConstraintSimilarity	*P1 and P2 have constraints that are known to be the same*	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 sem:sameConstraint ?c2)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	P1 and P2 have no constraints that are known to be the same	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 sem:differentConstraint ?c2)
ConnectedInstanceNamesSimilarityScore	*The names of the connected instances of P1 and P2 have a similarity score of 0.8 to 1.0*	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) similarString(?n1, ?n2, '0.8', '1.0', 'i')
	The names of the connected instances of P1 and P2 have a similarity score of 0.6 to 0.8	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) similarString(?n1, ?n2, '0.6', '0.8', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the connected instances of P1 and P2 have a similarity score of 0.4 to 0.6	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) similarString(?n1, ?n2, '0.4', '0.6', 'i')
	The names of the connected instances of P1 and P2 have a similarity score of 0.2 to 0.4	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) similarString(?n1, ?n2, '0.2', '0.4', 'i')
	The names of the connected instances of P1 and P2 have a similarity score of 0.0 to 0.2	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) similarString(?n1, ?n2, '0.0', '0.2', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
ConnectedInstanceNamesSynonymsOrEqual	*The names of the connected instances of P1 and P2 are either synonyms or equal*	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) synonyms(?n1, ?n2)
	The names of the connected instances of P1 and P2 are neither synonyms nor equal	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:value ?v1) (?c2 base:value ?v2) (?v1 base:name ?n1) (?v2 base:name ?n2) notSynonyms(?n1, ?n2)
SameRelationType	*The property kinds of P1 and P2 are the same*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) equal(?pt1, ?pt2)
	The property kinds of P1 and P2 are not the same	(?p1 base:type ?pt1) (?p2 base:type ?pt2) notEqual(?pt1, ?pt2)

Continued on next page

Table 11 (continued)

Random Variable	Target Space Values	Patterns
RelationTypeNamesSynonymsOrEqual	*The types of properties P1 and P2 have names that are synonyms or equal*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) synonyms(?pt1n, ?pt2n)
	The types of properties P1 and P2 have names that are not synonyms and not equal	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) notSynonyms(?pt1n, ?pt2n)
RelationTypeNamesSimilarityScore	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, '0.8', '1.0', 'i')
	The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, '0.6', '0.8', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, '0.4', '0.6', 'i')
	The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, '0.2', '0.4', 'i')
	The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:name ?pt1n) (?pt2 base:name ?pt2n) similarString(?pt1n, ?pt2n, '0.0', '0.2', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
RangeTypeNamesSynonymsOrEqual	*The names of the ranges of P1 and P2 are synonyms or equal*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) synonyms(?pt1rn, ?pt2rn)
	The names of the ranges of P1 and P2 are neither synonyms nor equal	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) notSynonyms(?pt1rn, ?pt2rn)
RangeTypeNamesSimilarityScore	*The names of the ranges of P1 and P2 have a similarity score of 0.8 to 1.0*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) similarString(?pt1rn, ?pt2rn, ’0.8’, ’1.0’, ’i’)

Continued on next page

Table 11 (continued)

Random Variable	Target Space Values	Patterns
	The names of the ranges of P1 and P2 have a similarity score of 0.6 to 0.8	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) similarString(?pt1rn, ?pt2rn, '0.6', '0.8', 'i')
	The names of the ranges of P1 and P2 have a similarity score of 0.4 to 0.6	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) similarString(?pt1rn, ?pt2rn, '0.4', '0.6', 'i')
	The names of the ranges of P1 and P2 have a similarity score of 0.2 to 0.4	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) similarString(?pt1rn, ?pt2rn, '0.2', '0.4', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the ranges of P1 and P2 have a similarity score of 0.0 to 0.2	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) (?pt1r base:name ?pt1rn) (?pt2r base:name ?pt2rn) similarString(?pt1rn, ?pt2rn, '0.0', '0.2', 'i')
SameRangeType	*The ranges of P1 and P2 are the same*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) equal(?pt1r, ?pt2r)
	The ranges of P1 and P2 are not the same	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:range ?pt1r) (?pt2 base:range ?pt2r) notEqual(?pt1r, ?pt2r)
DomainTypeNamesSynonymsOrEqual	*The names of the domains of P1 and P2 are synonyms or equal*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) synonyms(?pr1dn, ?pr2dn)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the domains of P1 and P2 are neither synonyms nor equal	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) notSynonyms(?pr1dn, ?pr2dn)
DomainTypeNamesSimilarityScore	*The names of the domains of P1 and P2 have a similarity score of 0.8 to 1.0*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) similarString(?pr1dn, ?pr2dn, '0.8', '1.0', 'i')
	The names of the domains of P1 and P2 have a similarity score of 0.6 to 0.8	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) similarString(?pr1dn, ?pr2dn, '0.6', '0.8', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	The names of the domains of P1 and P2 have a similarity score of 0.4 to 0.6	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) similarString(?pr1dn, ?pr2dn, '0.4', '0.6', 'i')
	The names of the domains of P1 and P2 have a similarity score of 0.2 to 0.4	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) similarString(?pr1dn, ?pr2dn, '0.2', '0.4', 'i')
	The names of the domains of P1 and P2 have a similarity score of 0.0 to 0.2	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) (?pr1d base:name ?pr1dn) (?pr2d base:name ?pr2dn) similarString(?pr1dn, ?pr2dn, '0.0', '0.2', 'i')

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
SameDomainType	*The domains of P1 and P2 are the same*	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) equal(?pr1d, ?pr2d)
	The domains of P1 and P2 are not the same	(?p1 base:type ?pt1) (?p2 base:type ?pt2) (?pt1 base:domain ?pr1d) (?pt2 base:domain ?pr2d) notEqual(?pr1d, ?pr2d)
DerivedConstraintSimilarity	*Both constraints have unit types and values defined and both are the same*	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 rdf:type base:DerivedConstraint) (?c2 rdf:type base:DerivedConstraint) (?c1 base:unitType ?u1) (?c2 base:unitType ?u2) (?c1 base:value ?v1) (?c2 base:value ?v2) equal(?u1, ?u2) equal(?v1, ?v2)

Continued on next page

Table 11 (continued)

Random Variable	**Target Space Values**	**Patterns**
	Both constraints have unit types and values defined and the values are not the same	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 rdf:type base:DerivedConstraint) (?c2 rdf:type base:DerivedConstraint) (?c1 base:unitType ?u1) (?c2 base:unitType ?u2) (?c1 base:value ?v1) (?c2 base:value ?v2) equal(?u1, ?u2) notEqual(?v1, ?v2)
	One of the constraints does not have a unit type defined	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) (?c1 base:unitType ?u1) noValue(?c2 base:unitType ?u2)
	Both constraints do not have a unit types defined	(?p1 base:constrainedBy ?c1) (?p2 base:constrainedBy ?c2) noValue(?c1 base:unitType ?u1) noValue(?c2 base:unitType ?u2)

A.2.2 Network Parameters

Table 12 documents the beliefs on the Bayesian network parameters, elicited as *Dirichlet* distributions (see section 2.3). The beliefs were elicited as probability distributions, and as a willingness to bet on an event (see section 7.2.4). The following is a sample elicitation question used in the process:

> *"Say the outcome of an experiment is a randomly selected pair of properties* $(P1, P2)$*, for which it is known that they are of a different type, have no constraints that are known to be semantically equivalent, and their owning entities* X *and* Y *are equivalent. How much are you prepared to stake (in fractions of \$1) in a gamble where you win \$1 if the properties* $P1$ *and* $P2$ *are semantically equivalent, and lose your stake if they are not (i.e., win \$0)?"*

To capture the belief as a probability distribution, *Dirichlet* distributions are used. *Beta* and *Dirichlet* distributions are particularly well suited for eliciting beliefs due to the natural way of specifying the parameters [152]. Note that a two-parameter Dirichlet distribution is equivalent to a Beta distribution.

Note that this Bayesian network illustrates the advantages of Bayesian networks explained in section 2.3 very well, in that (granting the validity of the independence assumptions, of course) only 167 beliefs need to be elicited as probability distributions to fully define the joint probability distribution. This is opposed to having to fill out a table with all possible combinations of values for the random variables, which would have 150,994,944 entries.

Table 12: Elicited beliefs on the network parameters of the comprehensive Bayesian network used for evaluating the approach.

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have similar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.92, 0.08)
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have similar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.8, 0.2)
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have similar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.85, 0.15)
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have similar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.2, 0.8)
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have dissimilar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.7, 0.3)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have dissimilar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.2, 0.8)
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have dissimilar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.18, 0.82)
AreSimilarParentEntities	*The property kinds of P1 and P2 are similar, Entities X and Y have dissimilar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.01, 0.99)
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have similar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.6, 0.4)
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have similar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.4, 0.6)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have similar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.55, 0.45)
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have similar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.02, 0.98)
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have dissimilar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.35, 0.65)
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have dissimilar types, P1 and P2 have constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(0.05, 0.95)
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have dissimilar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have similar names*	Dir(0.5, 0.5)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
AreSimilarParentEntities	*The property kinds of P1 and P2 are dissimilar, Entities X and Y have dissimilar types, P1 and P2 have no constraints that are known to be the same, Entities X and Y have dissimilar names*	Dir(1.0E-4, 0.9999)
AreEquivalentProperties	*The property kinds of P1 and P2 are similar, Entities X and Y which are parents of Properties P1 and P2 are equivalent, P1 and P2 have constraints that are known to be the same*	Dir(0.999, 0.001)
AreEquivalentProperties	*The property kinds of P1 and P2 are similar, Entities X and Y which are parents of Properties P1 and P2 are equivalent, P1 and P2 have no constraints that are known to be the same*	Dir(0.7, 0.3)
AreEquivalentProperties	*The property kinds of P1 and P2 are similar, Entities X and Y which are parents of Properties P1 and P2 are not equivalent, P1 and P2 have constraints that are known to be the same*	Dir(0.4, 0.6)
AreEquivalentProperties	*The property kinds of P1 and P2 are similar, Entities X and Y which are parents of Properties P1 and P2 are not equivalent, P1 and P2 have no constraints that are known to be the same*	Dir(0.02, 0.98)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
AreEquivalentProperties	*The property kinds of P1 and P2 are dissimilar, Entities X and Y which are parents of Properties P1 and P2 are equivalent, P1 and P2 have constraints that are known to be the same*	Dir(0.2, 0.8)
AreEquivalentProperties	*The property kinds of P1 and P2 are dissimilar, Entities X and Y which are parents of Properties P1 and P2 are equivalent, P1 and P2 have no constraints that are known to be the same*	Dir(0.005, 0.995)
AreEquivalentProperties	*The property kinds of P1 and P2 are dissimilar, Entities X and Y which are parents of Properties P1 and P2 are not equivalent, P1 and P2 have constraints that are known to be the same*	Dir(0.008, 0.992)
AreEquivalentProperties	*The property kinds of P1 and P2 are dissimilar, Entities X and Y which are parents of Properties P1 and P2 are not equivalent, P1 and P2 have no constraints that are known to be the same*	Dir(1.0E-4, 0.9999)
AreInconsistentProperties	*Properties P1 and P2 are equivalent, P1 and P2 have constraints that are known to be the same*	Dir(1.0E-4, 0.9999)
AreInconsistentProperties	*Properties P1 and P2 are equivalent, P1 and P2 have no constraints that are known to be the same*	Dir(0.999, 0.001)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
AreInconsistentProperties	*Properties P1 and P2 are not equivalent, P1 and P2 have constraints that are known to be the same*	Dir(1.0E-5, 0.99999)
AreInconsistentProperties	*Properties P1 and P2 are not equivalent, P1 and P2 have no constraints that are known to be the same*	Dir(1.0E-6, 0.999999)
EntityNamesSimilar	*Entities X and Y have names that are known synonyms or are equal, The names of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.999999, 1.0E-6)
EntityNamesSimilar	*Entities X and Y have names that are known synonyms or are equal, The names of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.9999, 1.0E-4)
EntityNamesSimilar	*Entities X and Y have names that are known synonyms or are equal, The names of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.99, 0.01)
EntityNamesSimilar	*Entities X and Y have names that are known synonyms or are equal, The names of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.98, 0.02)
EntityNamesSimilar	*Entities X and Y have names that are known synonyms or are equal, The names of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.93, 0.07)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
EntityNamesSimilar	*Entities X and Y have names that are not known synonyms and are not equal, The names of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.98, 0.02)
EntityNamesSimilar	*Entities X and Y have names that are not known synonyms and are not equal, The names of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.92, 0.08)
EntityNamesSimilar	*Entities X and Y have names that are not known synonyms and are not equal, The names of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.85, 0.15)
EntityNamesSimilar	*Entities X and Y have names that are not known synonyms and are not equal, The names of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.5, 0.5)
EntityNamesSimilar	*Entities X and Y have names that are not known synonyms and are not equal, The names of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.3, 0.7)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
EntityTypesSimilar	*The property kinds of P1 and P2 are similar*, *Entities X and Y have the same types*, *Entities X and Y have types with similar names*	Dir(0.999, 0.001)
EntityTypesSimilar	*The property kinds of P1 and P2 are similar*, *Entities X and Y have the same types*, *Entities X and Y have types with dissimilar names*	Dir(0.99, 0.01)
EntityTypesSimilar	*The property kinds of P1 and P2 are similar*, *Entities X and Y do not have the same types*, *Entities X and Y have types with similar names*	Dir(0.9, 0.1)
EntityTypesSimilar	*The property kinds of P1 and P2 are similar*, *Entities X and Y do not have the same types*, *Entities X and Y have types with dissimilar names*	Dir(0.4, 0.6)
EntityTypesSimilar	*The property kinds of P1 and P2 are dissimilar*, *Entities X and Y have the same types*, *Entities X and Y have types with similar names*	Dir(0.8, 0.2)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
EntityTypesSimilar	*The property kinds of P1 and P2 are dissimilar*, *Entities X and Y have the same types*, *Entities X and Y have types with dissimilar names*	Dir(0.99, 0.01)
EntityTypesSimilar	*The property kinds of P1 and P2 are dissimilar*, *Entities X and Y do not have the same types*, *Entities X and Y have types with similar names*	Dir(0.5, 0.5)
EntityTypesSimilar	*The property kinds of P1 and P2 are dissimilar*, *Entities X and Y do not have the same types*, *Entities X and Y have types with dissimilar names*	Dir(0.01, 0.99)
EntityTypeNamesSimilar	*Entities X and Y have types with synonymous or equal names*, *The names of the types of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.999999, 1.0E-6)
EntityTypeNamesSimilar	*Entities X and Y have types with synonymous or equal names*, *The names of the types of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.9999, 1.0E-4)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
EntityTypeNamesSimilar	*Entities X and Y have types with synonymous or equal names*, *The names of the types of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.99, 0.01)
EntityTypeNamesSimilar	*Entities X and Y have types with synonymous or equal names*, *The names of the types of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.98, 0.02)
EntityTypeNamesSimilar	*Entities X and Y have types with synonymous or equal names*, *The names of the types of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.93, 0.07)
EntityTypeNamesSimilar	*Entities X and Y have types with non synonymous and non equal names*, *The names of the types of X and Y have a similarity score of 0.8 to 1.0*	Dir(0.98, 0.02)
EntityTypeNamesSimilar	*Entities X and Y have types with non synonymous and non equal names*, *The names of the types of X and Y have a similarity score of 0.6 to 0.8*	Dir(0.92, 0.08)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
EntityTypeNamesSimilar	*Entities X and Y have types with non synonymous and non equal names, The names of the types of X and Y have a similarity score of 0.4 to 0.6*	Dir(0.85, 0.15)
EntityTypeNamesSimilar	*Entities X and Y have types with non synonymous and non equal names, The names of the types of X and Y have a similarity score of 0.2 to 0.4*	Dir(0.5, 0.5)
EntityTypeNamesSimilar	*Entities X and Y have types with non synonymous and non equal names, The names of the types of X and Y have a similarity score of 0.0 to 0.2*	Dir(0.3, 0.7)
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar, The types of properties P1 and P2 have similar names, The domains of P1 and P2 are similar, The property kinds of P1 and P2 are the same*	Dir(0.9999, 1.0E-4)
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar, The types of properties P1 and P2 have similar names, The domains of P1 and P2 are similar, The property kinds of P1 and P2 are not the same*	Dir(0.85, 0.15)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar*, *The types of properties P1 and P2 have similar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are the same*	Dir(0.1, 0.9)
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar*, *The types of properties P1 and P2 have similar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are not the same*	Dir(0.05, 0.95)
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are similar*, *The property kinds of P1 and P2 are the same*	Dir(0.4, 0.6)
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are similar*, *The property kinds of P1 and P2 are not the same*	Dir(0.02, 0.98)
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are the same*	Dir(0.01, 0.99)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
RelationKindsAreSimilar	*The ranges of P1 and P2 are similar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are not the same*	Dir(0.005, 0.995)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have similar names*, *The domains of P1 and P2 are similar*, *The property kinds of P1 and P2 are the same*	Dir(0.5, 0.5)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have similar names*, *The domains of P1 and P2 are similar*, *The property kinds of P1 and P2 are not the same*	Dir(0.4, 0.6)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have similar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are the same*	Dir(0.2, 0.8)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have similar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are not the same*	Dir(0.15, 0.85)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are similar*, *The property kinds of P1 and P2 are the same*	Dir(0.2, 0.8)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are similar*, *The property kinds of P1 and P2 are not the same*	Dir(0.001, 0.999)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are the same*	Dir(0.001, 0.999)
RelationKindsAreSimilar	*The ranges of P1 and P2 are dissimilar*, *The types of properties P1 and P2 have dissimilar names*, *The domains of P1 and P2 are dissimilar*, *The property kinds of P1 and P2 are not the same*	Dir(1.0E-4, 0.9999)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0*, *The types of properties P1 and P2 have names that are synonyms or equal*	Dir(0.999999, 1.0E-6)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.8 to 1.0*, *The types of properties P1 and P2 have names that are not synonyms and not equal*	Dir(0.98, 0.02)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8*, *The types of properties P1 and P2 have names that are synonyms or equal*	Dir(0.9999, 1.0E-4)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.6 to 0.8*, *The types of properties P1 and P2 have names that are not synonyms and not equal*	Dir(0.92, 0.08)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6*, *The types of properties P1 and P2 have names that are synonyms or equal*	Dir(0.99, 0.01)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.4 to 0.6*, *The types of properties P1 and P2 have names that are not synonyms and not equal*	Dir(0.85, 0.15)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4*, *The types of properties P1 and P2 have names that are synonyms or equal*	Dir(0.98, 0.02)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.2 to 0.4*, *The types of properties P1 and P2 have names that are not synonyms and not equal*	Dir(0.5, 0.5)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2*, *The types of properties P1 and P2 have names that are synonyms or equal*	Dir(0.93, 0.07)
RelationTypeNamesSimilar	*The names of the types of P1 and P2 have a similarity score of 0.0 to 0.2*, *The types of properties P1 and P2 have names that are not synonyms and not equal*	Dir(0.3, 0.7)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal*, *The names of the domains of P1 and P2 have a similarity score of 0.8 to 1.0*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal*, *The names of the domains of P1 and P2 have a similarity score of 0.8 to 1.0*, *The domains of P1 and P2 are not the same*	Dir(0.99, 0.01)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal*, *The names of the domains of P1 and P2 have a similarity score of 0.6 to 0.8*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal*, *The names of the domains of P1 and P2 have a similarity score of 0.6 to 0.8*, *The domains of P1 and P2 are not the same*	Dir(0.95, 0.05)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal*, *The names of the domains of P1 and P2 have a similarity score of 0.4 to 0.6*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal*, *The names of the domains of P1 and P2 have a similarity score of 0.4 to 0.6*, *The domains of P1 and P2 are not the same*	Dir(0.9, 0.1)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal, The names of the domains of P1 and P2 have a similarity score of 0.2 to 0.4, The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal, The names of the domains of P1 and P2 have a similarity score of 0.2 to 0.4, The domains of P1 and P2 are not the same*	Dir(0.85, 0.15)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal, The names of the domains of P1 and P2 have a similarity score of 0.0 to 0.2, The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are synonyms or equal, The names of the domains of P1 and P2 have a similarity score of 0.0 to 0.2, The domains of P1 and P2 are not the same*	Dir(0.7, 0.3)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal, The names of the domains of P1 and P2 have a similarity score of 0.8 to 1.0, The domains of P1 and P2 are the same*	Dir(0.999, 0.001)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.8 to 1.0*, *The domains of P1 and P2 are not the same*	Dir(0.98, 0.02)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.6 to 0.8*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.6 to 0.8*, *The domains of P1 and P2 are not the same*	Dir(0.8, 0.2)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.4 to 0.6*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.4 to 0.6*, *The domains of P1 and P2 are not the same*	Dir(0.5, 0.5)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.2 to 0.4*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.2 to 0.4*, *The domains of P1 and P2 are not the same*	Dir(0.3, 0.7)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.0 to 0.2*, *The domains of P1 and P2 are the same*	Dir(0.999, 0.001)
HaveSimilarDomain	*The names of the domains of P1 and P2 are neither synonyms nor equal*, *The names of the domains of P1 and P2 have a similarity score of 0.0 to 0.2*, *The domains of P1 and P2 are not the same*	Dir(0.1, 0.9)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.8 to 1.0*, *The ranges of P1 and P2 are the same*, *The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.999, 0.001)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.8 to 1.0*, *The ranges of P1 and P2 are the same*, *The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.8 to 1.0*, *The ranges of P1 and P2 are not the same*, *The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.99, 0.01)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.8 to 1.0*, *The ranges of P1 and P2 are not the same*, *The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.95, 0.05)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.6 to 0.8*, *The ranges of P1 and P2 are the same*, *The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.6 to 0.8*, *The ranges of P1 and P2 are the same*, *The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.999, 0.001)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.6 to 0.8*, *The ranges of P1 and P2 are not the same*, *The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.94, 0.06)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.6 to 0.8*, *The ranges of P1 and P2 are not the same*, *The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.8, 0.2)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.4 to 0.6*, *The ranges of P1 and P2 are the same*, *The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.4 to 0.6*, *The ranges of P1 and P2 are the same*, *The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.4 to 0.6*, *The ranges of P1 and P2 are not the same*, *The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.95, 0.05)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.4 to 0.6, The ranges of P1 and P2 are not the same, The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.7, 0.3)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.2 to 0.4, The ranges of P1 and P2 are the same, The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.2 to 0.4, The ranges of P1 and P2 are the same, The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.2 to 0.4, The ranges of P1 and P2 are not the same, The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.85, 0.15)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.2 to 0.4, The ranges of P1 and P2 are not the same, The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.4, 0.6)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.0 to 0.2, The ranges of P1 and P2 are the same, The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.0 to 0.2, The ranges of P1 and P2 are the same, The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.999, 0.001)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.0 to 0.2, The ranges of P1 and P2 are not the same, The names of the ranges of P1 and P2 are synonyms or equal*	Dir(0.7, 0.3)
HaveSimilarRange	*The names of the ranges of P1 and P2 have a similarity score of 0.0 to 0.2, The ranges of P1 and P2 are not the same, The names of the ranges of P1 and P2 are neither synonyms nor equal*	Dir(0.1, 0.9)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar, The names of the connected instances of P1 and P2 are either synonyms or equal, The names of the connected instances of P1 and P2 have a similarity score of 0.8 to 1.0*	Dir(0.98, 0.02)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
SimilarConnectedInstances	*The ranges of P1 and P2 are similar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.6 to 0.8*	Dir(0.88, 0.12)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.4 to 0.6*	Dir(0.85, 0.15)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.2 to 0.4*	Dir(0.78, 0.22)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.0 to 0.2*	Dir(0.7, 0.3)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
SimilarConnectedInstances	*The ranges of P1 and P2 are similar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.8 to 1.0*	Dir(0.97, 0.03)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.6 to 0.8*	Dir(0.8, 0.2)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.4 to 0.6*	Dir(0.5, 0.5)
SimilarConnectedInstances	*The ranges of P1 and P2 are similar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.2 to 0.4*	Dir(0.2, 0.8)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
SimilarConnectedInstances	*The ranges of P1 and P2 are similar*, *The names of the connected instances of P1 and P2 are neither synonyms nor equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.0 to 0.2*	Dir(0.05, 0.95)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.8 to 1.0*	Dir(0.5, 0.5)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.6 to 0.8*	Dir(0.45, 0.55)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.4 to 0.6*	Dir(0.35, 0.65)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.2 to 0.4*	Dir(0.3, 0.7)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are either synonyms or equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.0 to 0.2*	Dir(0.2, 0.8)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are neither synonyms nor equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.8 to 1.0*	Dir(0.4, 0.6)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar*, *The names of the connected instances of P1 and P2 are neither synonyms nor equal*, *The names of the connected instances of P1 and P2 have a similarity score of 0.6 to 0.8*	Dir(0.15, 0.85)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.4 to 0.6*	Dir(0.1, 0.9)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.2 to 0.4*	Dir(0.01, 0.99)
SimilarConnectedInstances	*The ranges of P1 and P2 are dissimilar, The names of the connected instances of P1 and P2 are neither synonyms nor equal, The names of the connected instances of P1 and P2 have a similarity score of 0.0 to 0.2*	Dir(1.0E-4, 0.9999)
EntityNamesSynonymsOrEqual	None	Dir(4.0E-4, 0.9996)
EntityNamesSimilarityScore	None	Dir(1.0E-4, 0.005, 0.1, 0.3, 0.5949)
EntityTypeNamesSynonymousOrEqual	None	Dir(0.02, 0.98)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
EntityTypeNamesSimilarityScore	None	Dir(0.001, 0.1, 0.2, 0.3, 0.399)
EntityTypesSame	None	Dir(0.02, 0.98)
ConstraintSimilarity	*Both constraints have unit types and values defined and both are the same*, *The connected instances through P1 and P2 are similar*	Dir(0.9999, 1.0E-4)
ConstraintSimilarity	*Both constraints have unit types and values defined and both are the same*, *The connected instances through P1 and P2 are not similar*	Dir(0.999, 0.001)
ConstraintSimilarity	*Both constraints have unit types and values defined and the values are not the same*, *The connected instances through P1 and P2 are similar*	Dir(0.01, 0.99)
ConstraintSimilarity	*Both constraints have unit types and values defined and the values are not the same*, *The connected instances through P1 and P2 are not similar*	Dir(0.001, 0.999)
ConstraintSimilarity	*One of the constraints does not have a unit type defined*, *The connected instances through P1 and P2 are similar*	Dir(0.65, 0.35)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
ConstraintSimilarity	*One of the constraints does not have a unit type defined, The connected instances through P1 and P2 are not similar*	Dir(0.001, 0.999)
ConstraintSimilarity	*Both constraints do not have a unit types defined, The connected instances through P1 and P2 are similar*	Dir(0.99, 0.01)
ConstraintSimilarity	*Both constraints do not have a unit types defined, The connected instances through P1 and P2 are not similar*	Dir(0.005, 0.995)
ConnectedInstanceNamesSimilarityScore	None	Dir(1.0E-4, 0.005, 0.1, 0.3, 0.5949)
ConnectedInstanceNamesSynonymsOrEqual	None	Dir(4.0E-4, 0.9996)
SameRelationType	None	Dir(0.05, 0.95)
RelationTypeNamesSynonymsOrEqual	None	Dir(0.05, 0.95)
RelationTypeNamesSimilarityScore	None	Dir(0.02, 0.1, 0.28, 0.35, 0.25)
RangeTypeNamesSynonymsOrEqual	None	Dir(0.02, 0.98)
RangeTypeNamesSimilarityScore	None	Dir(0.001, 0.1, 0.2, 0.3, 0.399)

Continued on next page

Table 12 (continued)

Random Variable	**Parent Values**	**Distribution**
SameRangeType	None	Dir(0.1, 0.9)
DomainTypeNamesSynonymsOrEqual	None	Dir(0.02, 0.98)
DomainTypeNamesSimilarityScore	None	Dir(0.02, 0.1, 0.28, 0.35, 0.25)
SameDomainType	None	Dir(0.001, 0.999)
DerivedConstraintSimilarity	None	Dir(1.0, 1.0, 1.0, 1.0)

A.3 Deterministic Classifiers: SPARQL Queries

A.3.1 Detection of Semantic Overlaps (Identification of TPs)

The following SPARQL query is used in determining the number of TPs for a deterministic classification of semantic overlaps. Note that line 9 is used in checking whether the equivalence of the two properties bound to **?p1** and **?p2** identified using the pattern specified in lines 10 to 22 is also contained in the list of actual equivalences.

```
 1  PREFIX overlap: <http://www.mbsec.gatech.edu/incon/ns/overlap#>
 2  PREFIX incon: <http://www.mbsec.gatech.edu/incon/ns/incon#>
 3  PREFIX base: <http://www.mbsec.gatech.edu/ns/base-structure#>
 4
 5  SELECT (COUNT(*) as ?numDistinctPairs)
 6  WHERE {
 7          SELECT DISTINCT ?p1 ?p2
 8          WHERE {
 9                  ?p1 overlap:equivalentToActual ?p2 .
10                  ?p1 a base:Property .
11                  ?p2 a base:Property .
12                  FILTER(?p1 != ?p2)
13                  ?x1 base:contains ?p1 .
14                  ?x2 base:contains ?p2 .
15                  ?p1 a ?pt1 .
16                  ?p2 a ?pt2 .
17                  ?pt1 base:name ?propertyName1 .
18                  ?pt2 base:name ?propertyName2 .
19                  ?x1 base:name ?parentName1 .
20                  ?x2 base:name ?parentName2 .
21                  FILTER(str(?propertyName1) = str(?propertyName2))
22                  FILTER(str(?parentName1) = str(?parentName2))
23          }
24  }
```

A.3.2 Detection of Semantic Overlaps (Identification of FPs)

The following SPARQL query is used in determining the number of FPs for a deterministic classification of semantic overlaps. The pattern is similar to that specified in the previous listing. The condition of the matching pair of properties being a false positive is ensured using the expressions in lines 22 to 27, which checks explicitly for *non*-containment in the list of introduced semantic equivalences.

```
 1  PREFIX overlap: <http://www.mbsec.gatech.edu/incon/ns/overlap#>
 2  PREFIX incon: <http://www.mbsec.gatech.edu/incon/ns/incon#>
 3  PREFIX base: <http://www.mbsec.gatech.edu/ns/base-structure#>
 4
 5  SELECT (COUNT(*) as ?numDistinctPairs)
 6  WHERE {
 7          SELECT DISTINCT ?p1 ?p2
 8          WHERE {
 9                  ?p1 a base:Property .
10                  ?p2 a base:Property .
11                  FILTER(?p1 != ?p2)
12                  ?x1 base:contains ?p1 .
13                  ?x2 base:contains ?p2 .
14                  ?p1 a ?pt1 .
15                  ?p2 a ?pt2 .
16                  ?pt1 base:name ?propertyName1 .
17                  ?pt2 base:name ?propertyName2 .
18                  ?x1 base:name ?parentName1 .
19                  ?x2 base:name ?parentName2 .
20                  FILTER(str(?propertyName1) = str(?propertyName2))
21                  FILTER(str(?parentName1) = str(?parentName2))
22                  FILTER NOT EXISTS {
23                          ?p1 overlap:equivalentToActual ?p2 .
24                  }
25                  FILTER NOT EXISTS {
```

```
            ?p2 overlap:equivalentToActual ?p1 .
        }
    }
}
```

A.3.3 Identification of Inconsistencies (Identification of TPs)

The following SPARQL query is used in determining the number of TPs for a deterministic classification of inconsistencies. Note that the pattern embeds the equivalence condition from the patterns in the previous listings (see lines 13 to 25). A check is performed whether the matching pattern is a part of the injected inconsistencies in lines 10 to 12.

```
1  PREFIX overlap: <http://www.mbsec.gatech.edu/incon/ns/overlap#>
2  PREFIX incon: <http://www.mbsec.gatech.edu/incon/ns/incon#>
3  PREFIX base: <http://www.mbsec.gatech.edu/ns/base-structure#>
4  PREFIX qudt: <http://qudt.org/schema/qudt#>
5
6  SELECT (COUNT(*) as ?numDistinctPairs)
7  WHERE {
8      SELECT DISTINCT ?p1 ?p2
9      WHERE {
10         ?i a incon:Inconsistency .
11         ?i incon:involves ?p1 .
12         ?i incon:involves ?p2 .
13         ?p1 a base:Property .
14         ?p2 a base:Property .
15         FILTER(?p1 != ?p2)
16         ?x1 base:contains ?p1 .
17         ?x2 base:contains ?p2 .
18         ?p1 a ?pt1 .
19         ?p2 a ?pt2 .
```

```
                ?pt1 base:name ?propertyName1 .
                ?pt2 base:name ?propertyName2 .
                ?x1 base:name ?parentName1 .
                ?x2 base:name ?parentName2 .
                FILTER(str(?propertyName1) = str(?propertyName2))
                FILTER(str(?parentName1) = str(?parentName2))
                ?p1 base:constrainedBy ?c1 .
                ?p2 base:constrainedBy ?c2 .
                {
                        ?c1 base:value ?v1 .
                        ?c2 base:value ?v2 .
                        FILTER(?v1 != ?v2)
                        FILTER NOT EXISTS
                        {
                                ?c1 base:unitType ?u1 .
                                ?c2 base:unitType ?u2 .
                        }
                }
                UNION
                {
                        ?c1 base:value ?v1 .
                        ?c2 base:value ?v2 .
                        FILTER(?v1 != ?v2)
                        ?c1 base:unitType ?u1 .
                        ?c2 base:unitType ?u2 .
                        FILTER(?u1 = ?u2)
                        ?u1 a qudt:SIBaseUnit .
                        ?u2 a qudt:SIBaseUnit .
                }
        }
}
```

A.3.4 Identification of Inconsistencies (Identification of FPs)

The following SPARQL query is used in determining the number of FPs for a deterministic classification of inconsistencies. The pattern is identical to that in the last listing, with the exception that the condition for a TP is removed, and a check is performed that the matching pair of properties is *not* in the list of injected inconsistencies (see lines 47 to 50).

```
 1  PREFIX overlap: <http://www.mbsec.gatech.edu/incon/ns/overlap#>
 2  PREFIX incon: <http://www.mbsec.gatech.edu/incon/ns/incon#>
 3  PREFIX base: <http://www.mbsec.gatech.edu/ns/base-structure#>
 4  PREFIX qudt: <http://qudt.org/schema/qudt#>
 5
 6  SELECT (COUNT(*) as ?numDistinctPairs)
 7  WHERE {
 8          SELECT DISTINCT ?p1 ?p2
 9          WHERE {
10                  ?p1 a base:Property .
11                  ?p2 a base:Property .
12                  FILTER(?p1 != ?p2)
13                  ?x1 base:contains ?p1 .
14                  ?x2 base:contains ?p2 .
15                  ?p1 a ?pt1 .
16                  ?p2 a ?pt2 .
17                  ?pt1 base:name ?propertyName1 .
18                  ?pt2 base:name ?propertyName2 .
19                  ?x1 base:name ?parentName1 .
20                  ?x2 base:name ?parentName2 .
21                  FILTER(str(?propertyName1) = str(?propertyName2))
22                  FILTER(str(?parentName1) = str(?parentName2))
23                  ?p1 base:constrainedBy ?c1 .
24                  ?p2 base:constrainedBy ?c2 .
25                  {
```

```
                        ?c1 base:value ?v1 .
                        ?c2 base:value ?v2 .
                        FILTER(?v1 != ?v2)
                        FILTER NOT EXISTS
                        {
                                ?c1 base:unitType ?u1 .
                                ?c2 base:unitType ?u2 .
                        }
                }
                UNION
                {
                        ?c1 base:value ?v1 .
                        ?c2 base:value ?v2 .
                        FILTER(?v1 != ?v2)
                        ?c1 base:unitType ?u1 .
                        ?c2 base:unitType ?u2 .
                        FILTER(?u1 = ?u2)
                        ?u1 a qudt:SIBaseUnit .
                        ?u2 a qudt:SIBaseUnit .
                }
                FILTER NOT EXISTS {
                        ?i a incon:Inconsistency .
                        ?i incon:involves ?p1 .
                        ?i incon:involves ?p2 .
                }
        }
}
```

REFERENCES

[1] Adams, J. B., "Probabilistic Reasoning and Certainty Factors," *Rule-Based Expert Systems*, pp. 263–271, 1984.

[2] Adourian, C. and Vangheluwe, H., "Consistency Between Geometric and Dynamic Views of a Mechanical System," in *Proceedings of the 2007 summer computer simulation conference*, p. 31, Society for Computer Simulation International, 2007.

[3] Alexander, C., Ishikawa, S., and Silverstein, M., "Pattern Languages," *Center for Environmental Structure*, vol. 2, 1977.

[4] Anderson, S. D., "Combining Evidence using Bayes' Rule," tech. rep., Wellesley College, 2007.

[5] Androutsopoulos, I., Koutsias, J., Chandrinos, K. V., and Spyropoulos, C. D., "An Experimental Comparison of Naive Bayesian and Keyword-based Anti-Spam Filtering with Personal E-Mail Messages," in *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 160–167, ACM, 2000.

[6] Anjorin, A., Lauder, M., Patzina, S., and Schürr, A., "eMoflon: Leveraging EMF and Professional CASE Tools," *Informatik*, p. 281, 2011.

[7] Baader, F., McGuinness, D., Nardi, D., and Patel-Schneider, P., *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[8] Balasubramanian, D., Narayanan, A., van Buskirk, C., and Karsai, G., "The Graph Rewriting and Transformation Language: GReAT," *Electronic Communications of the EASST*, vol. 1, 2007.

[9] Balogh, A. and Varró, D., "Advanced Model Transformation Language Constructs in the VIATRA2 Framework," in *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1280–1287, ACM, 2006.

[10] Balzer, R., "Tolerating Inconsistency," in *Proceedings of the 13th International Conference on Software Engineering*, pp. 158–165, IEEE, 1991.

[11] Barceló, P., Libkin, L., and Reutter, J. L., "Querying Graph Patterns," in *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ACM, 2011.

[12] Basole, R., Qamar, A., Park, H., Paredis, C., and McGinnis, L., "Visual Analytics for Early-Phase Complex Engineered System Design Support," 2015.

[13] Bastian, M., Heymann, S., Jacomy, M., and others, "Gephi: an Open Source Software for Exploring and Manipulating Networks," *ICWSM*, vol. 8, pp. 361–362, 2009.

[14] Berger, J. O., *Statistical Decision Theory and Bayesian Analysis.* Springer, 1985.

[15] Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., and Ökrös, A., "Incremental Evaluation of Model Queries over EMF Models," in *Model Driven Engineering Languages and Systems*, pp. 76–90, Springer, 2010.

[16] Bergmann, G., Ökrös, A., Ráth, I., Varró, D., and Varró, G., "Incremental Pattern Matching in the Viatra Model Transformation System," in *Proceedings of the Third International Workshop on Graph and Model Transformations*, pp. 25–32, ACM, 2008.

[17] Berlin, J. and Motro, A., "Database Schema Matching using Machine Learning with Feature Selection," in *Advanced information systems engineering*, pp. 452–466, Springer, 2002.

[18] Bernardo, J. M. and Smith, A. F., *Bayesian Theory*, vol. 405. John Wiley & Sons, 2009.

[19] Berners-Lee, T., Hendler, J., Lassila, O., and others, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[20] Blanc, X., Mougenot, A., Mounier, I., and Mens, T., "Incremental Detection of Model Inconsistencies Based on Model Operations," in *Advanced information systems engineering*, pp. 32–46, Springer, 2009.

[21] Boehm, B. and In, H., "Identifying Quality-Requirement Conflicts," *IEEE software*, vol. 13, no. 2, pp. 25–35, 1996.

[22] Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., and Houston, K. A., *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 3 ed., April 2007.

[23] Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T. F., and Weber, R., *The Design of Distributed Systems: an Introduction to Focus.* Citeseer, 1992.

[24] Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., and Ratiu, D., "Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 526–545, 2010.

[25] Broy, M. and Ştefănescu, G., "The Algebra of Stream Processing Functions," *Theoretical Computer Science*, vol. 258, no. 1, pp. 99–129, 2001.

[26] Bruegge, B. and Dutoit, A. H., *Object-Oriented Software Engineering Using UML, Patterns and Java.* Prentice Hall, 2004.

[27] Burkart, O., *Automatic Verification of Sequential Infinite-state Processes*, vol. 1354 of *Lecture Notes in Computer Science.* Springer, 1997.

[28] Carnap, R., "The Two Concepts of Probability: the Problem of Probability," *Philosophy and phenomenological research*, vol. 5, no. 4, pp. 513–532, 1945.

[29] Chang, C. C. and Keisler, H. J., *Model Theory.* Elsevier, 1990.

[30] Chapra, S. C. and Canale, R., *Numerical Methods for Engineers.* McGraw-Hill, 6 ed., 2009.

[31] Clarke, E. M. and Wing, J. M., "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[32] Cloutier, R. J. and Verma, D., "Applying the Concept of Patterns to Systems Architecture," *Systems engineering*, vol. 10, no. 2, pp. 138–154, 2007.

[33] Conte, D., Foggia, P., Sansone, C., and Vento, M., "Thirty Years of Graph Matching in Pattern Recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[34] Cooper, G. F., "The Computational Complexity of Probabilistic Inference using Bayesian Belief Networks," *Artificial intelligence*, vol. 42, no. 2, pp. 393–405, 1990.

[35] Csertdn, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., and Varró, D., "VIATRA – Visual Automated Transformations for Formal Verification and Validation of UML Models," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE)*, pp. 267–270, IEEE, 2002.

[36] Czarnecki, K. and Helsen, S., "Classification of Model Transformation Approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, pp. 1–17, 2003.

[37] Czarnecki, K. and Helsen, S., "Feature-based Survey of Model Transformation Approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[38] da Silva, M. A. A., Mougenot, A., Blanc, X., and Bendraou, R., "Towards Automated Inconsistency Handling in Design Models," in *Advanced Information Systems Engineering*, Springer, 2010.

[39] Dagum, P. and Luby, M., "An Optimal Approximation Algorithm for Bayesian Inference," *Artificial Intelligence*, vol. 93, no. 1, pp. 1–27, 1997.

[40] De Finetti, B., "Foresight: its Logical Laws in Subjective Sources," 1964.

[41] Dempster, A. P., "Upper and Lower Probabilities Induced by a Multivalued Mapping," *The annals of mathematical statistics*, pp. 325–339, 1967.

[42] Dempster, A. P., "A Generalization of Bayesian Inference," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 205–247, 1968.

[43] Dezert, J., Wang, P., and Tchamova, A., "On the Validity of Dempster-Shafer Theory," in *Information Fusion (FUSION), 2012 15th International Conference on*, pp. 655–660, IEEE, 2012.

[44] Diller, A., *Z: An Introduction to Formal Methods*, vol. 2. Wiley Chichester, UK, 1990.

[45] Doan, A., Domingos, P., and Halevy, A. Y., "Reconciling Schemas of Disparate Data Sources: A Machine-learning Approach," *SIGMOD Rec.*, vol. 30, pp. 509–520, May 2001.

[46] Douven, I., "Abduction," in *The Stanford Encyclopedia of Philosophy* (Zalta, E. N., ed.), spring 2011 ed., 2011.

[47] Drummond, C. and Holte, R. C., "Cost Curves: An Improved Method for Visualizing Classifier Performance," 2006.

[48] Easterbrook, S., "Handling Conflict between Domain Descriptions with Computer-Supported Negotiation," *Knowledge acquisition*, vol. 3, no. 3, pp. 255–289, 1991.

[49] Easterbrook, S. and Callahan, J., "Formal Methods for Verification and Validation of Partial Specifications: A Case Study," *Journal of Systems and Software*, vol. 40, no. 3, pp. 199–210, 1998.

[50] Easterbrook, S., Finkelstein, A., Kramer, J., and Nuseibeh, B., "Coordinating Distributed ViewPoints: the Anatomy of a Consistency Check," *Concurrent Engineering*, vol. 2, no. 3, pp. 209–222, 1994.

[51] Egyed, A., "Scalable Consistency Checking between Diagrams - The VIEWINTEGRA Approach," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pp. 387–390, IEEE, 2001.

[52] Egyed, A., "Instant Consistency Checking for the UML," in *Proceedings of the 28th international conference on Software engineering*, pp. 381–390, ACM, 2006.

[53] Egyed, A., "Fixing inconsistencies in UML design models," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 292–301, IEEE, 2007.

[54] Egyed, A., Letier, E., and Finkelstein, A., "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 99–108, IEEE, 2008.

[55] Egyed, A. and Medvidovic, N., "Consistent Architectural Refinement and Evolution using the Unified Modeling Language," in *1st Workshop on Describing Software Architecture with UML, co-located with ICSE*, pp. 83–87, Citeseer, 2001.

[56] Egyed, A. F., *Heterogeneous View Integration and its Automation.* PhD thesis, University of Southern California, 2000.

[57] Ehrig, H., Pfender, M., and Schneider, H. J., "Graph-Grammars: An Algebraic Approach," in *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pp. 167–180, IEEE, 1973.

[58] Ehrig, H., Prange, U., and Taentzer, G., "Fundamental Theory for Typed Attributed Graph Transformation," in *Graph transformations*, pp. 161–177, Springer, 2004.

[59] Ehrig, M., *Ontology Alignment: Bridging the Semantic Gap*, vol. 4. Springer Science & Business Media, 2006.

[60] Ehrig, M. and Sure, Y., "Ontology Mapping – An Integrated Approach," in *The Semantic Web: Research and Applications*, pp. 76–91, Springer, 2004.

[61] Eng, J., "Receiver Operating Characteristic Analysis: a Primer," *Academic radiology*, vol. 12, no. 7, pp. 909–916, 2005.

[62] Engels, G., Küster, J. M., Heckel, R., and Groenewegen, L., "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 186–195, ACM, 2001.

[63] Eppstein, D., "Subgraph Isomorphism in Planar Graphs and Related Problems," in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 1995.

[64] Evans, J., *The History and Practice of Ancient Astronomy.* Oxford University Press, 1998.

[65] Famelis, M., Salay, R., and Chechik, M., "Partial Models: Towards Modeling and Reasoning with Uncertainty," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 573–583, IEEE, 2012.

[66] Fawcett, T., "An Introduction to ROC Analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[67] Feldmann, S., Herzig, S. J. I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C. J. J., and Vogel-Heuser, B., "Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems," in *Proceedings of the 2015 IFAC Symposium on Information Control in Manufacturing (INCOM)*, (Ottawa, Canada), May 2015 (In Press).

[68] Finkelstein, A., "A Foolish Consistency: Technical Challenges in Consistency Management," in *DEXA*, Springer, 2000.

[69] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M., "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 01, pp. 31–57, 1992.

[70] Finkelstein, A., Spanoudakis, G., and Till, D., "Managing Interference," in *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints' 96) on SIGSOFT'96 Workshops*, ACM, 1996.

[71] Finkelstein, A. C., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B., "Inconsistency Handling in Multiperspective Specifications," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, 1994.

[72] Friedenthal, S., Moore, A., and Steiner, R., *A Practical Guide to SysML: the Systems Modeling Language*. 2011.

[73] Friedman-Hill, E., *Jess in Action*. Manning Greenwich, CT, 2003.

[74] Fritzson, P., *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[75] Fu, J., "Pattern Matching in Directed Graphs," in *Combinatorial Pattern Matching*, Springer, 1995.

[76] Gallagher, B., "Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching," *AAAI FS*, vol. 6, 2006.

[77] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[78] Gary, M. R. and Johnson, D. S., "Computers and Intractability: A Guide to the Theory of NP-completeness," 1979.

[79] Gausemeier, J., Giese, H., Schaefer, W., Axenath, B., Frank, U., Henkl;er, S., Pook, S., and Tichy, M., "Towards the Design of Self-Optimizing Mechatronic Systems: Consistency Between Domain-Spanning and Domain-Specific Models," in *Proceedings of the 16th International Conference on Engineering Design (ICED07)*, pp. 657–658, 2007.

[80] Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., and Rieke, J., "Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems," in *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*, vol. 6, 2009.

[81] Giarratano, J. C. and Riley, G., *Expert Systems.* PWS Publishing Co., 1998.

[82] Giese, H., Levendovszky, T., and Vangheluwe, H., "Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools," in *Models in Software Engineering*, Springer, 2007.

[83] Giese, H. and Wagner, R., "From Model Transformation to Incremental Bidirectional Model Synchronization," *Software & Systems Modeling*, vol. 8, no. 1, 2009.

[84] Gödel, K., "Über Formal Unentscheidbare Stze der Principia Mathematica und Verwandter Systeme I," *Monatshefte fr Mathematik*, vol. 38, pp. 173–198, 1931.

[85] Goyal, S. and Westenthaler, R., "RDF Gravity (RDF Graph Visualization Tool). Salzburg Research, Austria," 2013.

[86] Green, R., "Relationships in the Organization of Knowledge: an Overview," in *Relationships in the Organization of Knowledge*, pp. 3–18, Springer, 2001.

[87] Gross, J., Reichwein, A., Rudolph, S., Bock, D., and Laufer, R., "An Executable Unified Product Model Based on UML to Support Satellite Design," in *AIAA Space 2009 Conference & Exposition*, 2009.

[88] Grundy, J., "Determining the Cause of Design Model Inconsistencies," *Computer*, vol. 47, no. 8, 2014.

[89] Haarslev, V. and Müller, R., "RACER System Description," in *Automated Reasoning*, pp. 701–705, Springer, 2001.

[90] Hamming, R. W., "Error Detecting and Error Correcting Codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[91] Harel, D. and Rumpe, B., "Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff," 2000.

[92] Harel, D. and Rumpe, B., "Meaningful Modeling: What's the Semantics of "Semantics"?," *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[93] Harman, G. H., "The Inference to the Best Explanation," *The Philosophical Review*, pp. 88–95, 1965.

[94] Hazelrigg, G. A., "A Framework for Decision-Based Engineering Design," *Journal of Mechanical Design*, vol. 120, 1998.

[95] Hazelrigg, G. A., *Fundamentals of Decision Making for Engineering Design and Systems Engineering.* 2012.

[96] Hegedus, A., Horváth, A., Ráth, I., Branco, M. C., and Varró, D., "Quick fix generation for DSMLs," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 17–24, IEEE, 2011.

[97] Hehenberger, P., Egyed, A., and Zeman, K., "Consistency Checking of Mechatronic Design Models," in *Proceedings of IDETC/CIE*, 2010.

[98] Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G., "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 3, pp. 231–261, 1996.

[99] Helms, B., *Object-Oriented Graph Grammars for Computational Design Synthesis.* PhD thesis, Technische Universität München, 2013.

[100] Helms, B. and Shea, K., "Booggie – An Object-Oriented Graph Grammar Implementation for Engineering Design Synthesis," in *4th International Conference on Design Computing and CognitionDCC*, vol. 10, 2010.

[101] Herrlich, H. and Strecker, G. E., *Category Theory*, vol. 2. Allyn and Bacon Boston, 1973.

[102] Herzig, S., Qamar, A., Reichwein, A., and Paredis, C. J., "A Conceptual Framework for Consistency Management in Model-Based Systems Engineering," in *Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2011.

[103] Herzig, S. J. I., "A Framework for Consistency Management in Model-Based Systems Engineering," diplomarbeit (dipl.-ing.), Technische Universität München, Institute of Astronautics, 2011.

[104] Herzig, S. J. I. and Brandstaetter, M., "Applying Software Engineering Methodologies to Model-Based Systems Engineering," in *4th International Workshop on Systems & Concurrent Engineering for Space Applications (SECESA 2010)*, 2010.

[105] Herzig, S. J., Kruse, B., Ciccozzi, F., Denil, J., Salay, R., and Varró, D., "Towards an Approach for Orchestrating Design Space Exploration Problems to Fix Multi-Paradigm Inconsistencies," in *8th International Workshop on Multi-Paradigm Modeling MPM 2014*, p. 61, 2014.

[106] Herzig, S. J. and Paredis, C. J., "A Conceptual Basis for Inconsistency Management in Model-Based Systems Engineering," *Procedia CIRP*, vol. 21, pp. 52–57, 2014.

[107] Herzig, S. J. and Paredis, C. J., "Bayesian Reasoning Over Models," in *Workshop on Model Driven Engineering, Verification and Validation*, pp. 69–78, 2014.

[108] Herzig, S. J., Qamar, A., and Paredis, C. J., "An Approach to Identifying Inconsistencies in Model-Based Systems Engineering," *Procedia Computer Science*, vol. 28, pp. 354–362, 2014.

[109] Herzig, S. J., Rouquette, N. F., Forrest, S., and Jenkins, J. S., "Integrating Analytical Models with Descriptive System Models: Implementation of the OMG SyML Standard for the Tool-specific Case of MapleSim and MagicDraw," *Procedia Computer Science*, vol. 16, pp. 118–127, 2013.

[110] Hoare, C. A. R. and others, *Communicating Sequential Processes*, vol. 178. Prentice-hall Englewood Cliffs, 1985.

[111] Hodgson, R., Keller, P. J., Hodges, J., and Spivak, J., "QUDT – Quantities, Units, Dimensions and Data Types Ontologies." http://www.qudt.org, March 2014.

[112] Hoffman, R. R., Coffey, J. W., Carnot, M. J., and Novak, J. D., "An Empirical Comparison of Methods for Eliciting and Modeling Expert Knowledge," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 46, pp. 482–486, SAGE Publications, 2002.

[113] Hofstadter, D. R., *Gödel, Escher, Bach.* 1999.

[114] Holmes, G., Donkin, A., and Witten, I. H., "Weka: A Machine Learning Workbench," in *Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems*, pp. 357–361, IEEE, 1994.

[115] Hopcroft, J. E., Motwani, R., and Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation.* Pearson, 2007.

[116] Huber, F., Schätz, B., Schmidt, A., and Spies, K., "AutoFOCUS – a Tool for Distributed Systems Specification," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 467–470, Springer, 1996.

[117] Hunter, A. and Nuseibeh, B., “Managing Inconsistent Specifications: Reasoning, Analysis, and Action,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 4, pp. 335–367, 1998.

[118] Inc., M.-W., *Merriam-Webster's Collegiate Dictionary.* Merriam-Webster, 2004.

[119] International Council on Systems Engineering (INCOSE), *Systems Engineering Vision 2020.* International Council on Systems Engineering (INCOSE), September 2007.

[120] International Council on Systems Engineering (INCOSE), *Systems Engineering Handbook.* International Council on Systems Engineering (INCOSE), October 2011.

[121] ISO, “IEC 42010 Systems and Software Engineering: Architectural Description,” July 2007.

[122] Ives, Z. G., Halevy, A. Y., Mork, P., and Tatarinov, I., “Piazza: Mediation and Integration Infrastructure for Semantic Web Data,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 2, pp. 155–175, 2004.

[123] Jackson, M., “The Meaning of Requirements,” *Annals of Software Engineering*, vol. 3, no. 1, pp. 5–21, 1997.

[124] Japkowicz, N. and Shah, M., *Evaluating Learning Algorithms: a Classification Perspective.* Cambridge University Press, 2011.

[125] Jaynes, E. T., *Probability Theory: the Logic of Science.* Cambridge university press, 2003.

[126] Kahneman, D. and Tversky, A., “Prospect Theory: An Analysis of Decision Under Risk,” *Econometrica: Journal of the Econometric Society*, pp. 263–291, 1979.

[127] Kelly, J. C., “Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems Volume II: A Practitioner's Guide,” tech. rep., NASA, 1997.

[128] Kleene, S. C., *Mathematical Logic.* Courier Corporation, 2002.

[129] Kleene, S. C., de Bruijn, N., de Groot, J., and Zaanen, A. C., “Introduction to Metamathematics,” 1952.

[130] Klir, G. and Yuan, B., *Fuzzy Sets and Fuzzy Logic*, vol. 4. Prentice Hall New Jersey, 1995.

[131] Kolmogorov, A. N., *Foundations of the Theory of Probability.* Chelsea Publishing Co.

[132] Königs, A., "Model Transformation with Triple Graph Grammars," in *Model Transformations in Practice (Satellite Workshop of MODELS)*, p. 166, 2005.

[133] Larrosa, J. and Valiente, G., "Constraint Satisfaction Algorithms for Graph Pattern Matching," *Mathematical Structures in Computer Science*, vol. 12, no. 04, pp. 403–422, 2002.

[134] Lauritzen, S. L. and Spiegelhalter, D. J., "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 157–224, 1988.

[135] Laycock, H., "Object," in *The Stanford Encyclopedia of Philosophy* (Zalta, E. N., ed.), winter 2014 ed., 2014.

[136] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P., "The Generic Modeling Environment," in *Workshop on Intelligent Signal Processing*, vol. 17, (Budapest, Hungary), 2001.

[137] Levenshtein, V. I., "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

[138] Levesque, H. J. and Brachman, R. J., "Expressiveness and Tractability in Knowledge Representation and Reasoning," *Computational Intelligence*, vol. 3, no. 1, pp. 78–93, 1987.

[139] Lieberman, H. and Fry, C., "Will Software Ever Work?," *Communications of the ACM*, vol. 44, no. 3, pp. 122–124, 2001.

[140] Liew, A., "Understanding Data, Information, Knowledge and their Inter-Relationships," *Journal of Knowledge Management Practice*, vol. 8, no. 2, pp. 1–16, 2007.

[141] Lingas, A., "Subgraph Isomorphism for Biconnected Outerplanar Graphs in Cubic Time," *Theoretical Computer Science*, vol. 63, no. 3, 1989.

[142] Liu, W., Easterbrook, S., and Mylopoulos, J., "Rule-Based Detection of Inconsistency in UML Models," in *Workshop on Consistency Problems in UML-Based Software Development, Dresden, Germany*, 2002.

[143] Lynch, N. A. and Tuttle, M. R., "An Introduction to Input/Output Automata," 1988.

[144] MacGregor, R. and Bates, R., "The Loom Knowledge Representation Language," tech. rep., DTIC Document, 1987.

[145] McGinnis, L. and Ustun, V., "A Simple Example of SysML-Driven Simulation," in *Simulation Conference (WSC), Proceedings of the 2009 Winter*, pp. 1703–1710, IEEE, 2009.

[146] Mehlhorn, K. and Sanders, P., *Algorithms and Data Structures: The Basic Toolbox*. Springer Science & Business Media, 2008.

[147] Mens, T., Van Der Straeten, R., and DHondt, M., "Detecting and Resolving Model Inconsistencies using Transformation Dependency Analysis," in *Model Driven Engineering Languages and Systems*, Springer, 2006.

[148] Mens, T., Van Der Straeten, R., and Simmonds, J., "A Framework for Managing Consistency of Evolving UML Models," *Software Evolution with UML and XML*, 2005.

[149] Mitchell, T. M., *Machine Learning*. WCB McGraw-Hill, 1997.

[150] Myers, G. J., Sandler, C., and Badgett, T., *The Art of Software Testing*. John Wiley & Sons, 2011.

[151] NASA, "Report on Project Management in NASA: Phase II of the Mars Climate Orbiter Mishap Report," March 2000.

[152] Neapolitan, R. E., *Learning Bayesian Networks*, vol. 1. Prentice Hall Upper Saddle River, 2004.

[153] Neapolitan, R. E., *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. CreateSpace Independent Publishing Platform, 2012.

[154] Nickel, U., Niere, J., and Zündorf, A., "The FUJABA Environment," in *Proceedings of the 22nd international conference on Software engineering*, pp. 742–745, ACM, 2000.

[155] Nipkow, T., Paulson, L. C., and Wenzel, M., *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*, vol. 2283. Springer Science & Business Media, 2002.

[156] Nuseibeh, B., "Ariane 5: Who Dunnit?," *IEEE Software*, 1997.

[157] Nuseibeh, B. and Easterbrook, S., "The Process of Inconsistency Management: A Framework for Understanding," in *Proceedings of the Tenth International Workshop on Database and Expert Systems Applications*, IEEE, 1999.

[158] Nuseibeh, B., Easterbrook, S., and Russo, A., "Leveraging Inconsistency in Software Development," *Computer*, vol. 33, no. 4, 2000.

[159] Nuseibeh, B., Kramer, J., and Finkelstein, A., "A Framework for Expressing the Relationships between Multiple Views in Requirements Specification," *Software Engineering, IEEE Transactions on*, vol. 20, no. 10, pp. 760–773, 1994.

[160] Object Management Group (OMG), "Systems Modeling Language (SysML) Version 1.3." http://www.omg.org/spec/SysML/1.3/.

[161] Object Management Group (OMG), "MDA Guide Version 1.0.1." http://www.omg.org/mda/, June 2003.

[162] Object Management Group (OMG), "OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP." http://www.omg.org/spec/QVT/, 2003.

[163] Object Management Group (OMG), "Meta Object Facility (MOF) 2.0 Core Specification." http://www.omg.org/spec/MOF/, January 2006.

[164] Object Management Group (OMG), "Object Constraint Language (OCL) Version 2.3.1." http://www.omg.org/spec/OCL/2.4/, January 2012.

[165] Olderog, E.-R., *Nets, Terms and Formulas: Three Views of Concurrent Processes and their Relationship*, vol. 23. Cambridge University Press, 2005.

[166] OMG, "OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3," Tech. Rep. formal/2010-05-03, 2010.

[167] Owen, S., Anil, R., Dunning, T., and Friedman, E., *Mahout in Action.* Manning, 2011.

[168] Peak, R., Paredis, C., McGinnis, L., Friedenthal, S., and Burkhart, R., "Integrating System Design with Simulation and Analysis Using SysML," *INCOSE Insight Special Edition on MBSE*, vol. 12, no. 4, pp. 40–43, 2009.

[169] Pearl, J., *Bayesian Networks: A Model of Self-Activated Memory for Evidential Reasoning.* University of California (Los Angeles). Computer Science Department, 1985.

[170] Pearl, J., "Fusion, Propagation, and Structuring in Belief Networks," *Artificial intelligence*, vol. 29, no. 3, pp. 241–288, 1986.

[171] Princeton University, "About WordNet." http://wordnet.princeton.edu, 2010. WordNet.

[172] Qamar, A., *Model and Dependency Management in Mechatronic Design.* PhD thesis, KTH Royal Institute of Technology, 2013.

[173] Qamar, A., Herzig, S. J. I., Paredis, C. J. J., and Törngren, M., "Analyzing Semantic Relationships between Multi-formalism Models for Inconsistency Management," in *Proceedings of the IEEE Systems Conference*, (Vancouver, BC, Canada), April 2015 (In Press).

[174] Qamar, A. and Paredis, C., "Dependency Modeling and Model Management in Mechatronic Design," in *ASME 2012 Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2012.

[175] Qamar, A., Törngren, M., Wikander, J., and During, C., "Integrating Multi-Domain Models for the Design and Development of Mechatronic Systems," in *7th European Systems Engineering Conference EuSEC 2010, Stockholm, Sweden*, Stockholm, Sweden: INCOSE, 2010.

[176] Quillan, M. R., "Semantic Memory," in *Semantic Information Processing*, p. 227270, MIT Press, 1968.

[177] Ragnhild Van Der Straeten and Tom Mens and Stefan Van Baelen, "Challenges in Model-Driven Software Engineering," in *Models in Software Engineering*, pp. 35–47, Springer, 2009.

[178] Ramsey, F. P., "Truth and Probability," *The foundations of mathematics and other logical essays*, pp. 156–198, 1931.

[179] Reder, A. and Egyed, A., "Model/Analyzer: a Tool for Detecting, Visualizing and Fixing Design Errors in UML," in *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, pp. 347–348, ACM, 2010.

[180] Reder, A. and Egyed, A., "Determining the Cause of a Design Model Inconsistency," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1531–1548, 2013.

[181] Reichwein, A., *Application-Specific UML Profiles for Multidisciplinary Product Data Integration.* PhD thesis, Universität Stuttgart, 2011.

[182] Rogers, J. and Costello, M., "Smart Projectile State Estimation using Evidence Theory," *Journal of Guidance, Control, and Dynamics*, vol. 35, no. 3, pp. 824–833, 2012.

[183] Rouquette, N. and Jenkins, S., "OWL Ontologies and SysML Profiles: Knowledge Representation and Modeling," in *OMG Eclipse Symposium*, 2010.

[184] Rozenberg, G. and Ehrig, H., *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 1. World Scientific Singapore, 1997.

[185] Ryman, A. G., Le Hors, A. J., and Speicher, S., "OSLC Resource Shape: A Language for Defining Constraints on Linked Data," in *Linked Data on the Web (LDOW2013) Workshop*, May 2013.

[186] Saadatpanah, P., Famelis, M., Gorzny, J., Robinson, N., Chechik, M., and Salay, R., "Comparing the Effectiveness of Reasoning Formalisms for Partial Models," in *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, pp. 41–46, ACM, 2012.

[187] Savage, L. J., *The Foundations of Statistics.* Courier Corporation, 1972.

[188] Schatz, B., Braun, P., Huber, F., and Wisspeintner, A., "Consistency in Model-Based Development," in *Engineering of Computer-Based Systems*, IEEE, 2003.

[189] Schumann, J. M., *Automated Theorem Proving in Software Engineering.* Springer Science & Business Media, 2001.

[190] Shafer, G., *A Mathematical Theory of Evidence*, vol. 1. Princeton University Press, 1976.

[191] Shah, A. A., Kerzhner, A. A., Schaefer, D., and Paredis, C. J., "Multi-View Modeling to Support Embedded Systems Engineering in SysML," in *Graph Transformations and Model-Driven Engineering*, pp. 580–601, Springer, 2010.

[192] Shani, U., Wadler, D., and Wagner, M., "Engineering Model Mediation which Really Works," in *INCOSEIL 7th International Symposium*, pp. 4–5, 2013.

[193] Shore, B., "Systematic Biases and Culture in Project Failures," *Project Management Journal*, vol. 39, no. 4, pp. 5–16, 2008.

[194] Shortliffe, E., *Computer-based Medical Consultations: MYCIN.* Elsevier, 2012.

[195] Shortliffe, E. H. and Buchanan, B. G., "A Model of Inexact Reasoning in Medicine," *Mathematical biosciences*, vol. 23, no. 3, pp. 351–379, 1975.

[196] Shvachko, K., Kuang, H., Radia, S., and Chansler, R., "The Hadoop Distributed File System," in *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, IEEE, 2010.

[197] Simko, G., Levendovszky, T., Neema, S., Jackson, E., Bapty, T., Porter, J., and Sztipanovits, J., "Foundation for Model Integration: Semantic Backplane," in *Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE*, 2012.

[198] Smith, B. C., *Reflection and semantics in a procedural language.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1982.

[199] Spanoudakis, G. and Finkelstein, A., "Reconciling Requirements: a Method for Managing Interference, Inconsistency and Conflict," *Annals of Software Engineering*, vol. 3, no. 1, pp. 433–457, 1997.

[200] Spanoudakis, G. and Zisman, A., "Inconsistency Management in Software Engineering: Survey and Open Research Issues," *Handbook of Software Engineering and Knowledge Engineering*, vol. 1, 2001.

[201] Spivey, J. M., *Understanding Z: a Specification Language and its Formal Semantics.* No. 3, Cambridge University Press, 1988.

[202] Sprock, T. and McGinnis, L., "Towards automated access to supply chain analyses," in *Proceedings of the 2014 POMS Annual Conference*, 2014.

[203] Stefik, M., *Introduction to Knowledge Systems.* Morgan Kaufmann Publishers Inc., 1995.

[204] Stegmüller, W., *Das Wahrheitsproblem und die Idee der Semantik.* Springer, 1968.

[205] Stephenson, A. G., Mulville, D. R., Bauer, F. H., Dukeman, G. A., Norvig, P., LaPiana, L., Rutledge, P., Folta, D., and Sackheim, R., "Mars Climate Orbiter Mishap Investigation Board Phase I Report," 1999.

[206] Storey, N. R., *Safety Critical Computer Systems.* Addison-Wesley Longman Publishing Co., Inc., 1996.

[207] Storey, V. C., "Understanding Semantic Relationships," *The VLDB Journal,*, vol. 2, pp. 455–488, Oct. 1993.

[208] Sunetnanta, T. and Finkelstein, A., "Automated Consistency Checking for Multiperspective Software Specifications," in *Workshop on Advanced Separation of Concerns*, (Toronto, CA), 2001.

[209] Susman, G. I., *Integrating Design and Manufacturing for Competitive Advantage.* Oxford University Press New York, 1992.

[210] Sztipanovits, J. and Karsai, G., "Model-Integrated Computing," *Computer*, vol. 30, no. 4, pp. 110–111, 1997.

[211] Taentzer, G., *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, Technische Universitat Berlin, 1996.

[212] Talbott, W., "Bayesian Epistemology," in *The Stanford Encyclopedia of Philosophy* (Zalta, E. N., ed.), fall 2013 ed., 2013.

[213] Tammet, T., "Gandalf," *Journal of Automated Reasoning*, vol. 18, no. 2, pp. 199–204, 1997.

[214] Tarski, A., "The Semantic Conception of Truth: and the Foundations of Semantics," *Philosophy and phenomenological research*, vol. 4, no. 3, pp. 341–376, 1944.

[215] Theobald, M., Sozio, M., Suchanek, F., and Nakashole, N., "URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules," 2010.

[216] Thompson, S. C. and Paredis, C. J., "An Introduction to Rational Design Theory," in *ASME 2010 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. 59–72, American Society of Mechanical Engineers, 2010.

[217] Tversky, A. and Kahneman, D., "Rational Choice and the Framing of Decisions," *Journal of business*, pp. S251–S278, 1986.

[218] Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., and Varró, D., "EMF-IncQuery: An Integrated Development Environment for Live Model Queries," *Science of Computer Programming*, vol. 98, pp. 80–99, 2015.

[219] Ullmann, J. R., "An Algorithm for Subgraph Isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[220] Van Der Straeten, R., *Inconsistency Management in Model-Driven Engineering*. PhD thesis, PhD thesis, Vrije Universiteit Brussel, 2005.

[221] Van Der Straeten, R. and D'Hondt, M., "Model Refactorings through Rule-Based Inconsistency Resolution," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ACM, 2006.

[222] Van Der Straeten, R., Mens, T., Simmonds, J., and Jonckers, V., "Using Description Logic to Maintain Consistency between UML Models," in *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, Springer, 2003.

[223] van Rijsbergen, C. J., *Information Retrieval*. Butterworth-Heinemann, 2 ed., 1979.

[224] Vlastos, G., "Reasons and Causes in the Phaedo," *The Philosophical Review*, pp. 291–325, 1969.

[225] Vogel-Heuser, B., Legat, C., Folmer, J., and Feldmann, S., "Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit," Tech. Rep. TUM-AIS-TR-01-14-02, Technische Universität München, 2014.

[226] W3C, "SPARQL Protocol and RDF Query Language 1.1 Overview," 2013.

[227] W3C, "Resource Description Framework (RDF)," 2014.

[228] Walter, U., *Astronautics: the Physics of Space Flight*. John Wiley & Sons, 2012.

[229] Wang, P., "A Defect in Dempster-Shafer Theory," in *Proceedings of the Tenth international conference on Uncertainty in artificial intelligence*, pp. 560–566, Morgan Kaufmann Publishers Inc., 1994.

[230] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P., "SPASS Version 3.5," in *Automated Deduction–CADE-22*, pp. 140–145, Springer, 2009.

[231] West, D. B. and others, *Introduction to Graph Theory*, vol. 2. Prentice Hall Englewood Cliffs, 2001.

[232] Whitehead, A. N. and Russell, B., *Principia Mathematica*, vol. 2. University Press, 1912.

[233] Zadeh, L. A., "Review of: A Mathematical Theory of Evidence," *AI magazine*, vol. 5, no. 3, p. 81, 1984.

[234] Zhang, N. L. and Poole, D., "A Simple Approach to Bayesian Network Computations," in *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, 1994.

[235] Zhivich, M. and Cunningham, R. K., "The Real Cost of Software Errors," 2009.

VITA

Sebastian J. I. Herzig was born in Basel, Switzerland. He has an extensive international experience from living in Switzerland, Pakistan, South Africa, Germany and the United States of America. Sebastian holds a *Diplom Ingenieur* (Dipl.-Ing.) degree in Aerospace Engineering (the equivalent of a combined Bachelor of Science and Master of Science degree), and a Bachelor of Science in Computer Science, both from the Technische Universität München (TUM). Both degrees were awarded in 2011. After performing a part of his Diploma thesis (Master's thesis) at the Georgia Institute of Technology as a visiting student, he returned to pursue his doctorate under Dr. Chris Paredis.