MANAGING DESIGN-TIME UNCERTAINTY IN SOFTWARE MODELS

by

Michail Famelis

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2016 by Michail Famelis
Abstract

Managing Design-Time Uncertainty in Software Models

Michail Famelis
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

The concern for handling uncertainty in software pervades contemporary software engineering. A particular form of uncertainty is that which can exist at multiple stages of the development process, where developers are uncertain about the content of their artifacts. However, existing tools and methodologies do not support working in the presence of design-time uncertainty, i.e., uncertainty that developers have about the content of their artifacts in various stages of the development process, therefore having to mentally keep track of multitude of possible alternative designs. Because of this uncertainty, developers are forced to either refrain from using their tools until uncertainty is resolved, or to make provisional decisions and attempt to keep track of them in case they prove premature and need to be undone. These options lead to either under-utilization of resources or potentially costly re-engineering.

This thesis presents a way to avoid these pitfalls by managing uncertainty in a systematic way. We propose to work in the presence of uncertainty and to only resolve it when enough information is available. Development can therefore continue while avoiding premature design commitments.

In a pilot user study we found that, when asked to articulate design-time uncertainty in a free-form modelling scenario, people tend to explicate it within the software artifact itself, staying close to the existing notation. This lead us to adopt “partial models”, a formalism for representing sets of possible design alternatives while staying faithful to the underlying language. This way, the problem of managing uncertainty in software artifacts is turned into a problem of doing management of software artifacts that contain uncertainty. To manage partial models, we have thus leveraged several software engineering sub-disciplines to develop techniques for: (a) articulating uncertainty, (b) checking and enforcing properties, as well as generating appropriate user feedback, (c) applying transformations, and (d) systematically making decisions, as new information becomes available. The resulting partial model management framework utilizes novel abstraction and automation approaches and enables a principled and systematic approach to managing design-time uncertainty in the software development process.
To Natalie.
Acknowledgements

It has been an immense privilege and honour to apprentice with Marsha Chechik. Her mentorship, example, guidance, support and teaching has profoundly transformed me in more ways that one. From her I learned how to be a scientist, a writer, a researcher, an editor, a manager, a member of the academic community, a mentor, a planner, a communicator, a reviewer, a coordinator, and even a little bit of a salesman. Marsha has taught me to aspire to be demanding and fair, fierce and gentle, reliable and uncompromising, direct and subtle. Or, as Natalie put it, to aspire to “be the Marsha I want to see in the world”.

I have also been privileged to work closely with Rick Salay. He was always willing to put in the extra effort to explain difficult mathematical concepts to me, discuss my wildest research ideas and help me develop my skills in formal specification and reasoning. In some ways, I feel like Rick Salay has been almost like a co-supervisor to me.

I also thank Alessio Di Sandro for all the hard engineering work he put into Mu-MMINT. Without his titanic efforts much of what is described in this thesis would be just symbols on paper. Alessio has also been a great friend and I thoroughly enjoyed our chats about all the ways our continent of origin is suffering.

I am deeply grateful to people who gave me feedback and commented on my thesis. Specifically, I thank the people who were in my supervisory and examination committee: Steve Easterbrook, Renée Miller, Carlo Ghezzi, and Kelly Lyons. Their comments really helped clarify and solidify my thoughts about the concepts discussed in the thesis. The ideas and concepts in it would not be as clear without them. Also, over the years I received critical feedback which greatly helped me, for which I want to specifically thank: Reviewer #2 of the TOSEM manuscript Towards Modeling and Reasoning with Design-Time Uncertainty, Krzysztof Czarnecki, Benoit Baudry, Greg Wilson, Fabiano Dalpiaz, and Jordi Cabot.

I also wish to thank my collaborators and co-authors, whose contributions helped shape this thesis: Stephanie Santosa, Julia Rubin, Levi Lúcio, Naama Ben-David, Ahmed Mashiyat, Gehan Selim, Pooya Saadatpanah, Jan Gorzny, Shoham Ben-David, Nathan Robinson, James R Cordy, Juergen Dingel, Hans Vangheluwe, Ramesh S, Sahar Kokaly, Vivien Suen, Oloruntobi Ogumbiyi, Qiyu Zhu, and Kawin Ethayarajh.

I am also very lucky to have had excellent colleagues whose friendship, and camaraderie helped me grow as a person and as an academic. Aws Albarghouthi was a great friend and many times an inspiration for his efficiency and clarity of vision. Jocelyn Simmonds taught me how to get over the impostor syndrome and showed me how to navigate the academic space, building my confidence in my own capabilities in the process. Alicia Grubb taught me many things, most crucially to examine critically my own conduct and behaviour. I am also very grateful for the companionship, camaraderie and friendship of the many people I was lucky to meet and work side by side with through my years at U of T: Albert Angel, Anya Taflovich, Apostolos Dimitromanolakis, Benson Quach, Christina Christodoulakis, Elizabeth Patitsas, Foteini Agrafioti, George Anvrosiadis, Golnazo Elahi, Jennifer Horikoff, Jorge Aranda, Manos Papargelis, Maria Attarian, Maria Kyriakaki, Maria Pavlou, Marios Fokaeis, Michael Mathioudakis, Nan Niu, Neil Ernst, Nikos Anesiadis, Petros Spachos, Sotiris Liaskos, Tasos Zouzias, Yannis Prassas, Yannis Sakellaridis, Yannis Sarkas, Yi Li, Yuval Filmus, and Zach Cinkaid.

A doctorate is almost by definition a long and arduous process that has no guarantee of success. I am grateful to Steve Rutchinski and Marina Haloulos who made it administratively possible for me to
not drop out due to poor financial planning on my part. Additionally, I am thankful to Solomon Shapiro from the university's Counselling and Psychological Services, as well as the facilitators and participants at the B-Side workshop for helping me remain a balanced individual who is at peace with myself.

For their invaluable financial support that was so much needed at the very start of my studies, I am especially thankful to Kostas Stamatakis, Manolis Giakoumakis, and fr. Gavriil Tsafos. Kostas Stamatakis and Maria Pavlaki are responsible for the fact that, growing up, I always took for granted that I would do a PhD. However, as a fledgling undergraduate student, I would not have even made to graduate school if it was not for Kostas Kontogiannis, and Nikos Papaspyrou, professors at the National Technical University of Athens, who inspired and mentored me in crucial points of my intellectual development.

My time in graduate school coincides with the economic, social, and political crisis in Greece. I thus also want to especially thank the people who, either through the Greek Solidarity Initiative or the Greek Canadian Democratic Organization, helped me to constructively channel my concerns, worries, fears and hopes into meaningful political action: Dimitris Apostolopoulos, Nikos Gryspolakis, Yannis Angelopoulos, Tasos Dimopoulos, Platon Routis, Anna Kostouki, Dionysos Savopoulos, Antonis Batsos, and Penny Siganou.

In Rhythms of Resistance – Toronto, I found community, friendship, wisdom, strength, resilience, creativity, music, affection, acceptance, and an uncompromising desire to be better, more loving, more open, more accepting, more aware, more alive. I feel truly privileged to have played, marched, partied, argued, relaxed, sung, danced, eaten and co-conspired with all the RoR bandies: Amelia Murphy-Beaudoin, Andrea Albagli, Angela Woodcock, Anjali Helferty, Ayelit Gabler, Bonnie Briggs, Dave Campanella, Elizabeth Patitsas, Gary Pereira, Jamie Smith, Jesse Ovadia, Jonnie Bakan, Kelsy Vivash, Kuba Borowiec, Laura Severinac, Mary Lindsay, Melissa McKechnie, Natalie Mathieson, Olga Parysek, Sarah Michael, Steph McKechnie, Timothy Luchies, Yuval Filmus, Zak Smith, and Zena Lord. I am especially thankful and appreciative to the four wise women of the “power circle”: Katelyn Blascik, Laura Hartley, Cate Ahrens, and, of course, Natalie. They literally changed my life forever.

When I first came to Canada I did not expect to also find an adoptive family. The Caines and the Tuddenhams took me in me as their own. Annette, Michael, Barbara, Jean, Ardwell, Joyce, Matt, Alexandra, Sydney, Travis, Kara, Hannah, Jay, Liam, Jeff, Elizabeth, thank you for the warmth, acceptance, and affection with which you embraced me.

My siblings Marietta, Panagiota, and Panagiotis, thank you for all your love, support and understanding! Thank you for accepting me for myself and for being a lifeline to my roots. My parents Giorgos, Polymnia, thank you for ...everything. Thank you for the small things: for all the hours that you spent listening to me rambling on the phone, for cheering me on, for telling me news from home, for passing my greetings to friends and family, for sending me money when I needed it, for the household survival tips, for holding on to me: Dimitris Karagiannis once told me that listening to you talk about me felt as if I had never left Greece. And thank you for the big things: for my nurturing and upbringing, for freeing me to choose my path and giving me the space to pursue it. Thank you for trusting me with your hopes and your dreams. Thank you for all the sacrifices you made. As a parent, I now think I am beginning to understand just how hard my absence has been for you. I know you are very proud of me. I know you would be very proud of me regardless of how this adventure would have concluded.

Yuri, thank you for making everything urgent. You are my thesis baby and all of this would probably not be finished on time without you.
Most of all, I am thankful to and for Natalie Caine, my companion, accomplice, friend, partner and love. First of all, thank you for your patience. Thank you for your love and acceptance. Thank you for your emotional and intellectual support. Thank you for your strength and your wisdom. Thank you for making me see things in different ways, for opening my eyes to my own self, for educating and critiquing me. Thank you for your not-so-well-hidden labour: thank you for all the times you cooked for me, cleaned for me, tidied up for me, did my laundry, made the beds, did the shopping, kept track of things, organized our outings, kept the baby, made sure I was surrounded by the routine I needed to focus. Thank you for letting me pull all-nighters and thank you for letting me vegetate to recover from them. Thank you for insisting we go for walks: you were always right. Thank you for insisting I did not quit. Thank you for helping me quit smoking. Thank you for our child. This would not have been possible without you. Now it is my turn to nurture your dreams. At least, I guarantee you, you will never have to hear the dreaded phrase “I have to work on my thesis” ever again.
# Contents

1 Introduction ................................................................. 1
   1.1 Uncertainty in Software Development ......................... 1
   1.2 Mitigating Design-Time Uncertainty .......................... 5
   1.3 Scope of Design-Time Uncertainty Management ............. 7
   1.4 Challenges and Contributions ................................. 9
      1.4.1 Articulating uncertainty .............................. 12
      1.4.2 Performing automated verification ................... 13
      1.4.3 Applying transformations ............................ 15
      1.4.4 Making decisions ................................... 15
   1.5 Organization ........................................................ 17

2 Background .................................................................... 20
   2.1 Models and Metamodels ........................................... 20
      2.1.1 Modelling formalisms .................................. 20
      2.1.2 Propositional encoding of models ..................... 21
   2.2 Verification of Software Models ............................... 23
      2.2.1 Encoding of syntactic properties ..................... 24
      2.2.2 Verification of properties ............................. 24
   2.3 Model Transformations ............................................. 25
      2.3.1 Graph rewriting .......................................... 25
      2.3.2 Properties of transformations ......................... 27
   2.4 Model-Driven Architecture ...................................... 28
   2.5 Summary .................................................................. 29

3 Representing Uncertainty ................................................... 31
   3.1 Semantics .................................................................. 31
   3.2 Notation ..................................................................... 33
      3.2.1 Design rationale ......................................... 34
      3.2.2 Base language metamodel adaptation ............... 35
      3.2.3 Assessment ................................................... 38
   3.3 Reduced Forms ....................................................... 40
      3.3.1 Graphical Reduced Form (GRF) ......................... 42
      3.3.2 Propositional Reduced Form (PRF) .................... 43
   3.4 Constructing Partial Models ...................................... 44
List of Tables

1.1 Examples of concerns a developer might have when faced with design-time uncertainty. . . 4
2.1 \textit{atomToProposition}: mapping model atoms to propositional variables. ............. 22
4.1 Checking property $p$ on the partial model $M$ of type $T$. ................................. 52
4.2 Model size categories. ......................................................................................... 55
4.3 Categories of the size of the concretization set. .................................................... 55
4.4 Properties used for experimentation. ..................................................................... 56
4.5 Verification results per property. .......................................................................... 58
5.1 Results of applying the ORM rules to the Ecore metamodel. ................................. 72
7.1 Operator \textit{Construct} .......................................................................................... 87
7.2 Operator \textit{MakePartial} ....................................................................................... 87
7.3 Operator \textit{Expand} ............................................................................................... 88
7.4 Operator \textit{Transform} ......................................................................................... 88
7.5 Operator \textit{Verify} ................................................................................................. 89
7.6 Operator \textit{Deconstruct} ....................................................................................... 89
7.7 Operator \textit{Decide} ................................................................................................. 90
7.8 Operator \textit{Constrain} ......................................................................................... 90
7.9 Operator \textit{GenerateCounterExample} ................................................................. 91
7.10 Operator \textit{GenerateExample} ............................................................................ 91
7.11 Operator \textit{GenerateDiagnosticCore} ................................................................. 92
8.1 UMLet example details (cf. Tables 4.2 and 4.3). .................................................... 105
8.2 UMLet example observed speedups, compared with experimental observations. ...... 109
8.3 Design decisions in Petri net metamodels from the Atlantic Metamodel Zoo. Each column represents one metamodel. ................................................................. 111
D.1 Valid combinations of repair strategies for ClasslessInstance and DanglingOperation. . 150
List of Figures

1.1 (a) Initial PtPP model, created in the Development activity. (b) The PtPP model after going through the Evolution activity. ................................. 2

1.2 (a) Developing PtPP. Left: what is known, right: design decisions. (b) P2P model after making design decisions. (c) Simplified state machine metamodel. (d) “Fold Single Entry Action” refactoring transformation. (e) The PtPP model after refactoring. ................. 11

1.3 Partial model $M_{p2p}$ used to articulate the uncertainty in Figure 1.2(a). Top: graphical part of $M_{p2p}$. Bottom: propositional May formula for $M_{p2p}$. ........................................... 12

1.4 Semantics of the $M_{p2p}$ partial model in Figure 1.3: six alternative designs of the PtPP system. ................................................................. 14

1.5 (a-f) Versions of the concretizations in Figure 1.4 after being transformed using the FSEA transformation in Figure 1.2(d). .................................................. 16

1.6 Partial model encoding the transformed concretizations shown in Figure 1.5. ............... 17

1.7 The elements that reify the “compromise” alternative solution to the decision point “what happens when a download completes” are set to True and highlighted in green. Elements reifying other alternative solutions are set to False and greyed out. Maybe elements that are not part of this decision are left unaltered. ........................................... 18

2.1 Two example models, that are also concretizations of $M_{p2p}$, from Figure 1.2. .......... 21

2.2 Type graph and well-formedness constraints $\Phi_{SM}$ of a simplified metamodel used for defining State Machines. This type graph is a simplified version of the metamodel in Figure 1.2(c). ................................................................. 21

2.3 Representing attributes using typed graphs: (a) Class shown in the concrete UML syntax. (b) The same class shown as a typed graph. ................................. 22

2.4 (a) The graph union of the models in Figures 2.1(a,b). (b) An acyclic typed graph $m_1$. (c) An acyclic typed graph $m_2$. (d) The graph union of $m_1$ and $m_2$ is not acyclic. ......... 24

2.5 (a, b) Simplified UML Class Diagram models $m_a$ and $m_b$. (c) Model $m_c$ resulting from the application of $R_{EV}$ to $m_b$. (d) Simplified UML Class Diagram metamodel. (e) Transformation rule $R_{EV}$ for doing the Encapsulate Variable refactoring. ......................... 26

2.6 Parts of the rule $R_{EV}$. Each rule part contains only those elements whose label appears in bold serif font. ................................................................. 27

2.7 (a) Model C1. (b) Model C2, resulting from applying F1 to C1. (c) Model C3, resulting from applying F2 to C1. (d) Model C4, resulting from applying F2 to C3. (e) Transformation rule F1. (f) Transformation rule F2. (g) Transformation rule F3. (h) Model C5, resulting from applying F3 to C1. (i) Model C6, resulting from applying F3 to C1. ... 28
3.1 Example sketches produced by users when asked to model uncertainty in a modelling task. 34
3.2 (a) Simple Box metamodel $B$, (b) partialized Box metamodel $B_p$, (c) partial model, instance of $B_p$, that violates the multiplicity constraints of $B$. 36
3.3 (a) Base language in PtPP: simplified UML State Machine metamodel. (b) Partial State Machine metamodel. (c) Fragment from $M_{p2p}$. (d) A different way to model the same fragment that is expressible in the abstract syntax but not in the concrete syntax (May formula not shown). 37
3.4 Some equivalent partial models that encode the set $S_{\neg P2}$, consisting of the two concretizations in Figures 1.4(c, d): (a) in GRF (b) in PRF (c) containing two superfluous Maybe-annotated atoms ($Bt$ and $Ba$). 41
3.5 An example product line, representing a family of vending machines. The product line consists of a Feature Model (left) and a Featured Transition System (right). Elements of the transition system are annotated with presence conditions. Image source: [Classen et al., 2010]. 47
4.1 Checking properties with partial models (E1) compared to classically (E2). Speedup ($\frac{E2}{E1}$) versus model size for different degrees of uncertainty. 59
5.1 (a) partial model $M_1$, showing points of uncertainty. (b) $M_1$ as a typed graph (UML abstract syntax). (c-e) Concretizations $m_{11}, m_{12}, m_{13}$ of $M_1$. 64
5.2 Transformation rule $R_{EV}$ for doing the Encapsulate Variable refactoring. 65
5.3 (a-c) Classical models resulting from applying $R_{EV}$ to the classical models in Figure 5.1(c-e). (d) The result of applying the lifted version $R_{EV}$ to $M_1$ directly: the partial model $M_2$. 65
5.4 Applying the lifted version of $R_{EV}$ to the motivating example. Left: input model $M_1$, May graph and truth table of the May formula. Right: output model $M_2$, May graph and truth table of the May formula. 67
6.1 Construction of GRF based on Algorithm 3 (E4), compared to construction based on Algorithm 1 (E3): speedup versus model size for different degrees of uncertainty. 80
7.1 The Uncertainty Waterfall: an idealized timeline of uncertainty management. 84
7.2 Degree of uncertainty in different stages of the Uncertainty Waterfall. 84
7.3 The Extended Uncertainty Waterfall. 85
7.4 Uncertainty management operators overlayed on the Extended Uncertainty Waterfall. 92
7.5 Screenshot of Mu-Mmint. Left: interactive workspace showing different versions of PtPP. Middle: Graphical partial model $M_{p2p}$ for PtPP. Model elements that reify solutions in the uncertainty tree are annotated with (M). Right: Decision tree for $M_{p2p}$. 96
7.6 Highlighting the elements that reify the “compromise” alternative solution in Mu-Mmint. 97
7.7 Highlighting alternatives for the decision “What happens when a download is completed?” in Mu-Mmint. Maybe elements realizing the “benevolent” scenario are coloured in blue, those realizing the “selfish” scenario – in green, and those realizing the “compromise” scenario – in red. 98
7.8 Visualizing a counterexample to property $P_2$ in Mu-Mmint. 99
8.1 Sequence diagram fragment of the UMLet paste function, depicting the **execute** operation. The maintainer’s fix for bug #10 is encircled by a dashed line. 

8.2 Overview of the UMLet example, superimposed over the Uncertainty Waterfall model. 

8.3 Example Petri net token game. Left: **Transition t1** is enabled. Right: the Petri net after firing t1. 

8.4 Overview of the PTN metamodel example, superimposed over the Uncertainty Waterfall model. 

B.1 (a) SMT-LIB encoding of a randomly generated partial model $M^r$. (b) The partial model $M^r$. (c-d) Concretizations of $M^r$. (e) Configurations of the outgoing edges of node 1 that satisfy Property 1. Concretization (c) satisfies Property 1 whereas concretization (d) does not. 

C.1 Simplified class diagram metamodel. 
C.2 Relational database schema metamodel. 
C.3 Rules `classToTable`, `associationToTable`, and `attributeToColumn` used to perform the ORM transformation. 
C.4 Rules `attributeToForeignKey` and `associationToForeignKey` used to perform the ORM transformation. 
C.5 Partial Ecore metamodel, used as input to the ORM benchmark. 
C.6 May formula of the partial Ecore metamodel. 

D.1 **Mu-Mmint** model K0. 
D.2 Diagram of the **Mu-Mmint** partial model K1. 
D.3 Ground propositional encoding of the property U1. 
D.4 Ground propositional encoding of sequence diagram well-formedness constraints involved in checking the property U1. 
D.5 Ground propositional encoding of the property U2. 
D.6 Concretization K2 of the partial model K1, a counterexample demonstrating why checking U1 yields **Maybe**. **Mu-Mmint** has greyed out elements of K1 that are not also part of K2. 
D.7 Diagram of the partial model K3. 
D.8 Diagram of the final model K4, implementing the repairs RC4 and RD2. 
D.9 Diagram of the partial metamodel N0. 
D.10 Uncertainty tree of the partial metamodel N0. 
D.11 Ground propositional encoding of the property U3. 
D.12 Partial PTN metamodel N2. 
D.13 Partial PTN metamodel N3. The uncertainty tree is shown in Figure D.14. 
D.14 Uncertainty tree of the partial metamodel N3. 
D.15 Diagram of the final PTN metamodel N5.
# List of Algorithms

1. GRF Conversion ................................................................. 42
2. PRF Conversion ................................................................. 43
3. Construction of a partial model from a set of concrete models. 44
4. Valuation-to-model conversion ............................................ 54
5. Computing all concretizations of a given model that satisfy a given property. 77
Chapter 1

Introduction

“The reality of today’s software systems requires us to consider uncertainty as a first-class concern in the design, implementation, and deployment of those systems.”


1.1 Uncertainty in Software Development

Software engineers have created various development process models to describe how software is created. In some form or other, all major software development process models consist of four basic process activities: Specification, Development, Validation, and Evolution [Sommerville, 2007]. In the Specification activity, developers aim to understand and document the requirements and constraints of the system. In the Development activity, they use the requirements to produce an implementation of the system. In the Validation activity, also known as Verification and Validation, they confirm that the system meets both the requirements specified earlier and the expectations of its users. In the Evolution activity, they adapt the system to respond to changes in its requirements and maintain it by fixing any bugs that are discovered. While the four process activities are organized differently in each process model (e.g., sequentially in Waterfall, iteratively in Spiral, etc.), they help structure and contextualize the different engineering tasks performed by developers.

Consider the following toy example, loosely based on the BitTorrent protocol [Cohen, 2008], which we develop and use throughout the thesis. Assume that an engineering team is developing a protocol for peer-to-peer downloads, called PtPP. For simplicity, we assume they are using the Spiral process model. In the Specification activity, they identify the following requirements:

R1. The protocol should support three modes of operation: Idle, Leeching, and Seeding.

R2. While operating in the Leeching mode, the user is sharing and downloading an incomplete file.

R3. While in the Seeding mode, the user is sharing a complete file.

R4. While Idle, there is no download activity.
R5. While Idle, users can start a new download, thus switching to Leeching.

R6. It should always be possible to cancel sharing a file, returning to the Idle mode of operation.

In the Development activity, the team uses these requirements to architect an initial design of the protocol. This is shown in Figure 1.1(a), expressed as a state machine [Rumbaugh et al., 2004]. The model uses a simplified version of the state machine modelling language, which is defined using a meta-model [Atkinson and Kuhne, 2003], shown in Figure 1.2(c). Specifically, there is a state Idle, a state Leeching (sharing and downloading an incomplete file), and a state Seeding (sharing a complete file). When Leeching is initiated, the action \texttt{start()} is executed. In the Validation activity, the team checks their implementation against the requirements. They discover that requirement R6 is not satisfied. Therefore, in the following Evolution activity, they expand their design, producing the model shown in Figure 1.1(b). In it, sharing from the states Seeding and Leeching can be cancelled, triggering the action \texttt{cancel()}.

While the different process activities correspond to different stages in the software lifecycle, they all involve making decisions. In the PtPP example, in the Specification activity the team has to make decisions such as: \textit{What happens when a download is completed? Should users be able to restart downloads?} In Development they have to decide issues such as: \textit{In what platform should PtPP be implemented? Can the design be refactored to be more compact?} In Verification they have to address decisions such as: \textit{What are important properties that PtPP should satisfy? What verification method should be used?} And in Evolution, they must make decisions such as: \textit{Which requirements should be re-examined?} Each decision that the team makes shapes PtPP in specific ways, giving it some characteristics, capabilities or behaviours, while restricting others. At any point in its lifecycle, a software system is the accumulated result of answers to a myriad such questions about its every aspect.

In each development activity, developers may face a number of open decisions. However, they may lack sufficient information to resolve them. For example:

- In the Development stage, the team might discover that some of the requirements were unclear [van Lamsweerde, 2009]. For example, the combination of requirements R3 and R4 is unclear with respect to which mode PtPP should be in when there is only uploading activity. The team must therefore decide whether to refine the requirements or to allow uploading in the Idle state. How-
ever, setting up a new round of requirements elicitation with all the stakeholders may be non-trivial, causing delays, during which the team does not have enough information to proceed.

- In the Evolution stage, the team may have multiple ways to fix an inconsistency bug [Nentwich et al., 2003; Egyed et al., 2008]. For example, one strategy to make the PtPP model consistent with the requirements is to make cancel() an action that is triggered when the added transitions are made. Another strategy is to make cancel() an entry action of the Idle state. The latter solution avoids duplication but depends on finalizing the requirements for Idle. This is an example where delays on one “decision point” cascade to others.

- At any point during the development, the team might have to consolidate the views of multiple stakeholders [Sabetzadeh and Easterbrook, 2006]. For example, assume that the organization developing PtPP is structured so that the development team is different from the testing team. If at any point in the development cycle the specifications used by the two teams are inconsistent (e.g., between identifying the violation of requirement R6 and fixing it), the teams have to go through a negotiation process to consolidate them in a single specification. Before that process is completed, neither team has sufficient information about how PtPP should behave.

- The team may have to decide whether to provide a particular functionality as a runtime configuration option or whether it should be a point of variability [Chechik et al., 2016]. For example, assume that the PtPP team is instead developing a product line [Pohl et al., 2005] of protocols and are faced with the decision “Should the system allow users to restart downloads?” They can decide to make it a runtime configuration option available to the users of every product or they can opt to make it a feature [Kang et al., 1990] thus providing it by default for some products and excluding it completely from the rest. However, making this decision requires further requirements analysis and therefore it cannot be decided until the next iteration.

- An example of external factors impacting development is the decision about choosing an implementation platform for PtPP. This decision depends on non-technical parameters such as business analysis, contractual obligations, licensing, marketing strategies, etc. Until that decision is made by the organization developing PtPP, the development team cannot decide whether, for example, to use particular optimization techniques, platform-specific design patterns, etc.

The uncertainty faced by developers who do not have sufficient information to make decisions about their system is called design-time uncertainty [Ramirez et al., 2012]. Design-time uncertainty concerns the content of a software system, and is different from uncertainty about the environment in which the system is meant to operate (known as environmental uncertainty). In other words, it is uncertainty that the developer has about what the system should be like, rather than about what conditions it may face during its operation. For example, design-time uncertainty occurs if the team has insufficient information to choose an implementation platform for PtPP. In contrast, environmental uncertainty occurs if, for example, it is not possible to know whether there will be a network outage at the time when a user decides to restart a download. To better illustrate the notion of design-time uncertainty, we brainstormed various concerns that a developer faced with design-time uncertainty might have. The resulting (incomplete) list contains several examples of such concerns and is shown in Table 1.1.

To further clarify the notion of design-time uncertainty, we refer to the philosophical classification provided by [Esfahani and Malek, 2012], which describes uncertainty in terms of two axes: (a) reducible
Table 1.1: Examples of concerns a developer might have when faced with design-time uncertainty.

<table>
<thead>
<tr>
<th>#</th>
<th>Concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What are the open decisions about the design of the system at this time?</td>
</tr>
<tr>
<td>2</td>
<td>What criteria are relevant for making each decision?</td>
</tr>
<tr>
<td>3</td>
<td>How likely is it that the criteria for making decisions will change?</td>
</tr>
<tr>
<td>4</td>
<td>What are the candidate solutions for each open design decision?</td>
</tr>
<tr>
<td>5</td>
<td>What criteria are relevant for a design to be considered a candidate solution?</td>
</tr>
<tr>
<td>6</td>
<td>Is it possible to generate candidate solutions automatically?</td>
</tr>
<tr>
<td>7</td>
<td>Which candidate solutions have been automatically generated and which were created by hand?</td>
</tr>
<tr>
<td>8</td>
<td>How likely is it that the criteria for generating candidate solutions will change?</td>
</tr>
<tr>
<td>9</td>
<td>How is each candidate solution expressed in the system?</td>
</tr>
<tr>
<td>10</td>
<td>Which parts of the system are affected by open design decisions?</td>
</tr>
<tr>
<td>11</td>
<td>How are design decisions decomposed?</td>
</tr>
<tr>
<td>12</td>
<td>Which design decisions do I intend to further decompose?</td>
</tr>
<tr>
<td>13</td>
<td>What are dependencies between design decisions?</td>
</tr>
<tr>
<td>14</td>
<td>What are the dependencies between candidate solutions?</td>
</tr>
<tr>
<td>15</td>
<td>Have I described all dependencies between design decisions or between candidate solutions?</td>
</tr>
<tr>
<td>16</td>
<td>Given a design decision, is its set of candidate solutions complete?</td>
</tr>
<tr>
<td>17</td>
<td>Which design decisions am I still working on?</td>
</tr>
<tr>
<td>18</td>
<td>What are relevant metadata about candidate solutions? (E.g., pros and cons)</td>
</tr>
<tr>
<td>19</td>
<td>What are the trade-offs involved in making each design decision?</td>
</tr>
<tr>
<td>20</td>
<td>What are relevant metadata about design decisions? (E.g., importance, priority)</td>
</tr>
<tr>
<td>21</td>
<td>What is it that causes me to not be able to make a design decision?</td>
</tr>
<tr>
<td>22</td>
<td>What is the rationale for each design decision?</td>
</tr>
<tr>
<td>23</td>
<td>What is the rationale for each candidate solution?</td>
</tr>
<tr>
<td>24</td>
<td>What is the rationale for the way each candidate solution is expressed in the system?</td>
</tr>
<tr>
<td>25</td>
<td>Is the expression of each candidate solution in the system stable?</td>
</tr>
<tr>
<td>26</td>
<td>What are commonalities and differences between candidate solutions?</td>
</tr>
<tr>
<td>27</td>
<td>What parts of a given candidate solution am I still working on?</td>
</tr>
<tr>
<td>28</td>
<td>What properties should all candidate solutions satisfy?</td>
</tr>
<tr>
<td>29</td>
<td>What properties should be satisfiable by at least one candidate solution?</td>
</tr>
<tr>
<td>30</td>
<td>Which candidate solutions can be incorporated into the system using conditionals?</td>
</tr>
<tr>
<td>31</td>
<td>Which candidate solutions can become configuration options?</td>
</tr>
<tr>
<td>32</td>
<td>Which design decisions can become points of variability or adaptation?</td>
</tr>
<tr>
<td>33</td>
<td>Which parts of the system are intentionally underspecified?</td>
</tr>
<tr>
<td>34</td>
<td>Are the candidate solutions consistent/well-formed?</td>
</tr>
<tr>
<td>35</td>
<td>Are they at the right level of abstraction?</td>
</tr>
<tr>
<td>36</td>
<td>Who identified each design decision?</td>
</tr>
<tr>
<td>37</td>
<td>Who proposed each candidate solution?</td>
</tr>
<tr>
<td>38</td>
<td>Who can help me resolve a given design decision?</td>
</tr>
<tr>
<td>39</td>
<td>Who is responsible for making each design decision?</td>
</tr>
<tr>
<td>40</td>
<td>What are the design decisions that have been made about the system in the past?</td>
</tr>
<tr>
<td>41</td>
<td>What was the rationale for making them?</td>
</tr>
<tr>
<td>42</td>
<td>Who made them?</td>
</tr>
<tr>
<td>43</td>
<td>Which past design decisions are affected by open design decisions?</td>
</tr>
<tr>
<td>44</td>
<td>How are open design decisions affected by past decisions?</td>
</tr>
<tr>
<td>45</td>
<td>Are there candidate solutions that were discarded in the past that might help in the present?</td>
</tr>
<tr>
<td>46</td>
<td>Which design decisions were made provisionally in the past?</td>
</tr>
<tr>
<td>47</td>
<td>What engineering tasks can be done at the part of the system that is affected by uncertainty?</td>
</tr>
<tr>
<td>48</td>
<td>What engineering tasks can be done at the rest of the system?</td>
</tr>
</tbody>
</table>
versus irreducible, and (b) aleatory versus epistemic. As Esfahani and Malek put it, “aleatory and epistemic represent the essence of uncertainty, while irreducible and reducible represent the managerial aspect of uncertainty.” We further clarify these concepts below.

Irreducible uncertainty refers to phenomena that are inherently unknowable, i.e., where uncertainty persists even in the presence of complete information. Reducible uncertainty, on the other hand, refers to things that are knowable, potentially in the future, when complete information becomes available.

An example of irreducible uncertainty is the Heisenberg Uncertainty Principle in quantum mechanics. All examples of uncertainty that we have mentioned in the PtPP scenario are examples of reducible uncertainty: there is nothing in the laws of physics that precludes the development team from making each decision. Design-time uncertainty is always reducible. In other words, a designer that has complete information does not have any uncertainty about how to build a system.

Esfahani and Malek describe aleatory uncertainty as uncertainty that is “caused by randomness and is typically modelled with probabilities.” In contrast, epistemic uncertainty is caused by insufficient knowledge. In the PtPP example, aleatory uncertainty would occur if the team was using statistical models to predict, e.g., the network load of a seeding server. In contrast, the uncertainty regarding the choice of implementation platform is epistemic: the development team does not have the marketing information to make the decision, and thus has to depend on other departments of their organization. Esfahani and Malek clarify that the distinction between epistemic and aleatory uncertainty “is motivated by the location of the uncertainty — in the decision-maker or in the physical system.” Design-time uncertainty is uncertainty that a developer has as a decision-maker; therefore it is typically manifested as epistemic uncertainty. In other words, design-time uncertainty captures the developer’s lack of knowledge about how to make a design decision. According to Esfahani and Malek context is an important source of aleatory uncertainty. Since each design decision occurs in a particular context, randomness in that context also affects the developer’s uncertainty about the design decision. This is an aleatory version of design-time uncertainty, where randomness in the development environment impacts the underlying assumptions of a design decision and therefore the criteria for determining which solutions are acceptable. For example, while deciding on the choice of implementation platform for PtPP, the team might determine that some platform has a high probability of becoming obsolete in the short term. Including it to the set of acceptable solutions depends on the developers’ attitude towards risk.

1.2 Mitigating Design-Time Uncertainty

To tackle environmental uncertainty, developers use various strategies such as self-adaptation [Esfahani and Malek, 2012], probabilistic behaviour [Hinton et al., 2006], identifying and explicating operational assumptions [Goldsby and Cheng, 2010], etc. These mitigation strategies result in functional systems that can operate under uncertain conditions. Thus, mitigating environmental uncertainty entails creation of uncertainty-aware software. In contrast, design-time uncertainty cannot be “coded away” but must be taken into account in the process by which software is created. The reason for this is that existing software tools, languages and techniques assume that developers are able to make all relevant decisions, i.e., that their input does not contain any uncertainty. For example, model transformations [Czarnecki and Helsen, 2006] cannot be applied unless uncertainty is resolved. This renders design-time uncertainty an undesirable characteristic that needs to either be avoided or removed altogether before resuming development. In other words, mitigating design-time uncertainty entails creation of uncertainty-aware
Aleatory aspects of design-time uncertainty can be mitigated by techniques such as risk management [Islam and Houmb, 2010; Boehm, 1991] and software estimation [Jones, 1998]. However, in the face of epistemic design-time uncertainty, developers are forced to either:

(a) avoid working on the uncertain parts of the system, delaying the decisions as long as necessary,
(b) entirely remove uncertainty by making educated guesses based on experience so that work can continue, or
(c) fork and maintain sets of alternative solutions.

We discuss these options below.

There is a considerable body of engineering research that focuses on avoiding design-time uncertainty by delaying making decisions until the most opportune moment. This is an established practice in many engineering disciplines, such as in industrial and mechanical engineering. A well known example is the Toyota Production System (TPS) [Monden, 2011], which was developed in Japan during the mid-20th century with the aim to eliminate inefficiencies in the production of automotive vehicles. The TPS was a precursor to the practice of lean manufacturing, which in the software world inspired Lean Software Engineering [Poppendieck and Poppendieck, 2003a; Ladas, 2009], one of the tenets of the Agile methodology [Fowler and Highsmith, 2001; Martin, 2003]. While Agile is becoming increasingly popular, it is not (to use the phrase coined by Fred Brooks [Brooks, 1995]) a “silver bullet” that is appropriate in every organizational setting and project. This is evidenced by the proliferation of literature discussing success factors [Misra et al., 2009; Ramesh et al., 2006] and challenges to its adoption [Nerur et al., 2005]. Moreover, while the stated goal of lean methods is to eliminate waste, i.e., under-utilization of resources, that is not necessarily the case in practice. For example, Ikonen et al. conducted a seven-week empirical study to identify sources of waste in lean development [Ikonen et al., 2010], focusing specifically on the Kanban method [Ladas, 2009]. Among other sources of waste, they identified that there were delays due to some developers waiting, e.g., for the completion of tasks that were under-estimated, for clarification of requirements, or for customer validation. Thus, delaying decisions is not always the most effective way to handle design-time uncertainty.

In practice, developers often rely on experience and craftsmanship to make and keep track of provisional decisions that artificially remove uncertainty from their artifacts so that development can continue. This increases the risk of having to backtrack their work if new information shows that the provisional decisions were wrong. Even worse, it can mean committing too early to design decisions that cannot be reversed without significant costs, when it would be more desirable to keep many alternative options open for consideration. In fact, the skillful management of such provisionality is a defining characteristic of expert software designers [Petre, 2009]. Agile proposes to manage these risks by employing short iteration cycles and frequent customer feedback. As discussed earlier, Agile is not a “silver bullet”, so such practices are not always feasible. Additionally, there is the problem of keeping track of which decisions were made provisionally and which were not. This is aggravated by the preference in Agile for “working software over comprehensive documentation” [Fowler and Highsmith, 2001]. Unless explicit traceability and provenance is maintained, the provisional character of a decision may be forgotten, thus implicitly turning the decision into a premature, undocumented commitment.

Given a design decision, engineers might choose to consider all alternative solutions, therefore forking the project into parallel streams. This allows them to keep their options open, as well as to potentially
Chapter 1. Introduction

7

turn decisions into variability points, thus creating families of products that meet the needs of more than one customer [Pohl et al., 2005; Chechik et al., 2016]. Forking can be done in a vigorous and systematic way. For example, the “Programming by Optimization” (PbO) approach, developed by Holger Hoos [Hoos, 2012], aims to help software developers avoid premature optimization commitments in settings where multiple algorithms can accomplish the same computational task, albeit in different ways. Design decisions and alternatives are explicitly tracked until the time when designers have enough hard evidence (obtained systematically using machine learning) regarding the optimality of each algorithm to make informed decisions. However, forking does not scale as a generic solution to mitigating design-time uncertainty due to the combinatoric explosion of possibilities in the case where multiple decision points must be managed. Additionally, since every fork must be maintained separately, every forking approach is limited by the size of the set of possible solutions to a decision point. Without any structure to describe their commonalities and differences, it is impossible to reuse results across forks. Thus, every engineering task such as verification, transformation, evolution, etc. has to be repeated for each fork, which is expensive and error-prone.

Thesis Statement. In this thesis, we introduce a novel methodological approach for managing epistemic design-time uncertainty, that combines ideas from the aforementioned mitigation strategies. Specifically, we show that it is possible to defer design decisions while efficiently maintaining and working with the encoding of the set of possible designs they entail. Once enough information is available, decisions can be made by systematically reducing this set. Similar to techniques that delay decision making, our approach aims to help developers avoid premature decisions. However, instead of routing around the parts of the system that are uncertain, we expand them to the set of possible solutions. Unlike forking approaches, however, we use a special-purpose formalism, called “partial models”, to encode the set of solutions in a compact and exact manner. Partial models are first class development artifacts that can be used to perform engineering tasks. This allows engineering work to proceed but, in contrast to approaches that employ provisionality, no decision is made prematurely. Once enough information is acquired, it is systematically incorporated in partial models through a process of formal refinement. These characteristics of partial models allow us to create a methodology specifically for managing design-time uncertainty.

In the rest of the thesis, we often refer to epistemic design-time uncertainty as “design-time uncertainty” or simply as “uncertainty”, provided that the context is unambiguous.

1.3 Scope of Design-Time Uncertainty Management

The two core ideas of the approach presented in this thesis are the deferral of design decisions and the use of formal artifacts to explicitly capture what is and is not known. This combination is not new: it has been studied extensively in the field of behavioural specification [Larsen and Thomsen, 1988]. The novelty of our approach lies in expanding that work for arbitrary modelling languages, not just partial behavioural models with carefully defined semantics. In this section, we use concepts from epistemology and partial behavioural modelling to precisely define the scope of our approach for design-time uncertainty management.

The idea of creating formal artifacts that incorporate the developer’s lack of knowledge about some part of a software system is not new. By definition, formal specification languages are used to explicate
what is known about a system. The unknown is therefore often modelled indirectly, by omission. Specifically: when specifying a system, *scope*\(^1\) defines what elements are relevant to the system, and *span* – what level of abstraction is acceptable. In the PtPP example, concerns such as logging and digital rights management are outside the scope of the model. Implementation details such as establishing TCP/IP connections are outside its span.

In this thesis, we make an assumption that the *scope* and *span* are known to the developers. In other words, we do not discuss the process of identifying which parts of their system they are uncertain about and what the decision points are.

A common convention in behavioural modelling is that elements that are outside the *scope* and *span* of the specification are ones about which the developer does not *care*. A different interpretation is that such elements represent “unknown unknowns”, a term we borrow from the simple epistemology of ignorance famously defined by the former US Secretary of Defence, Donald Rumsfeld prior to the 2003 invasion of Iraq, during a Press Conference at the NATO Headquarters, in Brussels on June 6, 2002 [Rumsfeld, 2011]:

“[As] we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns the ones we don’t know we don’t know.”

Design-time uncertainty encompasses both “known unknowns” and “unknown unknowns”. Elements outside the *scope* and *span* of a specification (“unknown unknowns”) are those that result from decisions that have an unknown set of possible solutions. “Known unknowns” result from decisions that have a known set of possible answers.

For example, we assume that the team developing PtPP is uncertain about three decisions, listed on the right of Figure 1.2(a): (D1) *Can users initiate seeding?* (D2) *Can downloads be restarted?* (D3) *what policy is followed when leeching ends?* The first two questions are closed: they have a “yes” or “no” answer, and are thus “known unknowns”. The third question is open, as it does not imply a set of acceptable candidate policies. It is therefore an “unknown unknown”.

In the context of software development, the relationship between “unknown unknowns” and “known unknowns” can be understood as a process of elicitation. A design decision is first posed as an open question, the set of its possible solutions being unknown. As the engineers engage and work through the problem, a set of possible solutions emerges. The design decision thus becomes a closed question: “which of the possible solutions should be selected?” In our example, as the PtPP team starts working on decision (D3), they decide that there are three possible alternatives regarding the policy about completed downloads: (a) the “benevolent” option which automatically starts *Seeding*, (b) the “selfish” option described earlier, and (c) a “compromise” option where no new connections are accepted while waiting for existing peers to complete their copies. The decision therefore becomes a choice between these three candidate solutions.

In this thesis, we scope our treatment of design-time uncertainty to “known unknowns”. Specifically, we do not investigate the process by which a development team elicits candidate solutions for an open question. Instead, we assume that this process has already taken place and that for each design decision

---

\(^1\)In this section, the word “scope” is used both to refer to the scope of the approach presented in the thesis, as well as to the scope of a software specification. The meaning of the word is the same for both usages.

\(^2\)The philosopher Slavoj Zizek has noted that Rumsfeld’s epistemology is incomplete, since it does not include “unknown knowns”, i.e., unstated assumptions [Zizek, 2006].
we are given a finite set of possible solutions. We also assume that the development team has concluded how to implement each candidate solution. Partial models are then used to compactly and exactly encode the set of candidate solutions.

Given that for each design decision the set of candidate solutions is assumed to be known, a development team may decide to include all or some solutions to the software system, allowing any of them to be used as required. This can be done, for example, by making the design decision a conditional choice in code or a configuration option, or by making it a point of variability [Pohl et al., 2005] or runtime adaptation [Bencomo et al., 2008]. We consider this to be a particular way to resolve uncertainty, that effectively turns “known unknowns” into “known knowns”. In other words, in such cases uncertainty about the design is considered resolved, even if the system is allowed to change in response to environmental uncertainty or user preferences. In this thesis, we do not address the process required to implement such resolutions of design-time uncertainty.

A conventional way to reason about “known unknowns” is the closed-world assumption [Shepherdson, 1984] via the use of underspecification. Specifically, elements that are within the given scope and span, but are not present in the specification are underspecified. In practice, this is often used to indicate that these elements are to be re-visited at a later point when more information becomes available. Thus, an underspecified system represents a set of fully specified possibilities. Nondeterminism is a common way to interpret underspecification in the context of behavioural modelling. Underspecification is used to implicitly (through omission) hint at what developers are uncertain about. However, the aim of our approach is to support decision deferral while allowing developers to perform engineering tasks with the artifacts containing design-time uncertainty. Performing engineering tasks with such artifacts requires design-time uncertainty to be explicitly modelled; otherwise, it cannot be manipulated or reasoned with. Therefore, the strategy of modelling design-time uncertainty through underspecification is not suitable for our purposes.

We summarize the topics that are outside the scope of this thesis below:

- Identifying what parts of a software system a development team is uncertain about and what the relevant design decisions are.

- Eliciting the set of candidate solutions for each design decision and their implementation.

- Managing the aleatory aspects of design-time uncertainty, i.e., randomness in the development context that impacts the underlying assumptions of a design decision and therefore the criteria for determining which solutions are acceptable candidates.

- Turning decision points into conditionals, configuration options, or points of variability or runtime adaptation.

1.4 Challenges and Contributions

In this dissertation, we consider software development from the perspective of Model-Based Software Engineering (MBSE) [Beydeda et al., 2005], where software artifacts are abstracted using appropriate modelling languages. We thus propose [Famelis et al., 2011; Famelis, 2012] an approach for managing uncertainty in software models by doing model management [Bernstein, 2003] of models that contain uncertainty, i.e., partial models. Model management entails putting together a coherent collection
of operators that allow performing different model manipulation tasks. Our approach to uncertainty management is thus outlined in Chapter 7 as a coherent collection of software engineering tasks that can be accomplished using partial models. Specifically, we consider partial models to be first-class artifacts that allow developers to:

(a) articulate their uncertainty,

(b) defer its resolution, while continuing development with tasks such as:

(i) performing automated verification and diagnosis, as well as

(ii) applying model transformations, and finally

(c) systematically resolve uncertainty by incorporating new information.

To illustrate the meaning of each of the verification and transformation tasks, we first show how they can be applied to the PtPP example, without the use of partial models. We then use the PtPP example to describe the main challenges and the contributions of the thesis.

For each area of contribution, we also indicate which concerns from Table 1.1 the approach helps address. Concerns from Table 1.1 that are not referenced in this section, are outside the scope of the thesis, as discussed in the previous sections.

**Illustration of the PtPP scenario.** In our scenario, the PtPP developers encountered uncertainty regarding three decisions, listed on the right of Figure 1.2(a). Specifically, the team is uncertain about the following decisions: (D1) Can users initiate seeding? (D2) Can downloads be restarted? (D3) what policy is followed when leeching ends? The team further elicited three possible solutions to (D3): (a) a “benevolent” option which automatically starts Seeding, (b) a “selfish” option described earlier, and (c) a “compromise” option where no new connections are accepted while waiting for existing peers to complete their copies.

Assume that the development team decides that their preferred strategy for managing uncertainty is to make provisional decisions. In our scenario, the team makes provisional choices for the three design decisions, as shown in Figure 1.2(b). Specifically: 1. Users can initiate seeding (indicated by the transition with action share()). 2. Downloads are not restarted (no effect on the model). 3. The “selfish” policy is adopted: at the end of Leeching, PtPP goes directly to Idle, not letting other connected peers to complete their work (indicated by the transition with action completed()) .

In the course of developing PtPP, the team may do additional work on the model. Refactoring is a common task done during development. For example, the team could refactor PtPP with a transformation such as the rewriting rule “Fold Single Entry Action” (FSEA) shown in Figure 1.2(d). FSEA is expressed as a graph rewriting transformation rule [Ehrig et al., 2006] using the notation of the Henshin model transformation engine [Arendt et al., 2010]. The rule tries to match a State s that has only one incoming Transition t1. The “negative application condition” (indicated by \<< forbid >>) stops the refactoring from being applied if there exists a second incoming Transition t2. Otherwise, it deletes Action a1 of t1 (indicated by \<< delete >>) and adds a new EntryAction with the same name (indicated by \<< create >>), associating it with s. The result of refactoring PtPP with FSEA is shown in Figure 1.2(e), where the actions share() and start() have been respectively folded into the states Seeding and Leeching.
Chapter 1. Introduction

A) Can users initiate seeding?

B) Can users restart downloads?

C) What happens when a download is completed?

Figure 1.2: (a) Developing PtPP. Left: what is known, right: design decisions. (b) P2P model after making design decisions. (c) Simplified state machine metamodel. (d) “Fold Single Entry Action” refactoring transformation. (e) The PtPP model after refactoring.

After making design decisions and further modifying the model, the team may decide to perform some analysis on their model. In our example, they check a property $P_1$: “no two transitions enabled in the same state can lead to the same target state”. The PtPP model that the team has created after making design decisions and subsequent modifications (shown in Figure 1.2(e)) does not satisfy this property due to the two outgoing transitions from Leeching, with cancel() and completed() respectively. Assuming that the team considers $P_1$ to be an indispensable property for PtPP to have, the result of analysis means that they now must re-examine their design decisions and potentially undo some of them, along with all the work (e.g. refactoring) that was done since. In other words, some of the design decisions taken earlier proved to be premature.

In order to avoid these pitfalls, the PtPP developers decide to adopt our approach for managing
design time uncertainty. In the following, we illustrate the main challenges in supporting the PtPP team defer design decisions while still developing the protocol and the contributions made in this thesis to address them.

1.4.1 Articulating uncertainty

**Challenges** The main challenge is to create appropriate abstractions and representations to enable the efficient representation of uncertainty by developers in machine-processable software artifacts. There are thus two major sub-problems: (a) creating a user-friendly notation for expressing uncertainty, and (b) equipping the notation with formal semantics, so that automated techniques can be applied.

As discussed in Section 1.3, we assume that for each design decision, developers have elicited a finite set of candidate solutions. Thus, an additional challenge is to create an effective process for expressing the set as a partial model.

**Contributions** We address the first two challenges by using the approach illustrated below and discussed in detail in Chapter 3. With respect to the third challenge, we provide an algorithm for automatically constructing a partial model from the given set of solutions in Section 3.4. Additionally, the tool Mu-MMINT, presented in Chapter 7, provides an interactive editor for creating partial models. However,
Chapter 1. Introduction

since the tool is a proof-of-concept prototype, we do not attempt to study the effectiveness and usability of its editor as the main method for the creation of partial models.

At the start of the PtPP scenario, the team’s design may be separated into known and unknown parts, as shown in Figure 1.2(a). In our approach, this information is captured in a single partial model $M_{p2p}$, shown in Figure 1.3.

The top part of Figure 1.3 shows the graphical part of $M_{p2p}$. It consists of a model expressed in the same concrete syntax as the original PtPP model fragment, with the addition of annotations (represented as seven-pointed star icons) to some elements. These are called Maybe elements and are used to explicate points of uncertainty in the model. We use propositional variables as aliases for individual Maybe element decisions. For example, the transition $\text{share}()$ between Idle and Seeding has two seven-pointed star icons annotations, with the variables $\text{At}$ and $\text{Aa}$ for the transition and its action respectively. These indicate that the developer is unsure whether to include them in the model. For example, setting $\text{Aa}$ to True means that she decides to include the action $\text{share}()$, False to exclude it.

The bottom part of Figure 1.3 shows the propositional May formula of $M_{p2p}$. This formula is used to explicate dependencies between points of uncertainty. For example, it specifies that each transition co-occurs with its corresponding action. Additionally, it specifies what are allowable configurations of Maybe elements if uncertainty is resolved. According to the solutions elicited by the team, the May formula allows three possible solutions to the policy about completed downloads: “benevolent”, “selfish”, and “compromise”. As shown in Figure 1.2(a), a different point of uncertainty is whether users should be able to initiate seeding (i.e., having a transition $\text{share}$ from Idle). The May formula in Figure 1.3, expresses the team’s decision to correlate the two points of uncertainty, by allowing the ability to start seeding only for the “selfish” and the “compromise” behavioural scenarios.

The model $M_{p2p}$ compactly encodes the set of models representing all possible ways to resolve uncertainty. Each such model is called a concretization. For illustration purposes, we show the complete set of concretizations of the $M_{p2p}$ partial model in Figure 1.4(a-f). For example, the model in Figure 1.2(b) is the same as the concretization shown in Figure 1.4(c). Moreover, the models in Figure 1.4(a,c,e) implement the “benevolent”, “selfish” and “compromise” designs, respectively. The models in Figure 1.4(b,d,f) augment the previous ones with restarts.

This approach helps address concerns 1, 4, 9, 10, 11, 13, 14, 26, and 35 from Table 1.1, by allowing developers to express their uncertainty within the model of the software system, as well as the different dependencies between the different concretizations.

1.4.2 Performing automated verification

Challenges Our first task is to identify the kinds of properties that can be verified in the presence of uncertainty. Subsequently, we need to develop appropriate verification techniques to efficiently check such properties. We also must determine what useful feedback we can provide to developers.

Contributions We address these challenges by using the approach illustrated below and detailed in Chapter 4. Using partial models, we can check syntactic properties, such as the property $P_1$ in the PtPP example (“no two transitions enabled in the same state can lead to the same target state”). The objective of such checking is to determine whether a property $\Phi$ holds for all, some, or none of the partial model’s concretizations. The result is therefore not boolean; instead, it can be either True (“the property holds...
Figure 1.4: Semantics of the $M_{g2p}$ partial model in Figure 1.3: six alternative designs of the PtPP system.

For all concretizations), False (“it does not hold for any concretization”), or Maybe (“it only holds for some”). For the last two cases, it is desirable to present users with a counterexample concretization.

For example, checking $P_1$ for $M_{g2p}$ results in Maybe, since the property holds for some concretizations (the ones in Figures 1.4(a,b,e,f)) but not for others (the ones in Figures 1.4(c,d)). Given this result, we can help the developer decide what to do next by generating appropriate feedback, such as showing one of the counter-examples, or creating a partial model encoding all the counter-examples. For example,
they may chose to enforce the property $P_1$, by restricting the possible concretizations of $M_{p2p}$ to the subset that satisfies it.

The checking of properties is accomplished using a technique that we introduced in [Fanelis et al., 2011]. To check a property $\Phi$, the May formula and the model’s well-formedness constraints (together called the model’s “Propositional Reduced Form” – PRF) are combined with $\Phi$ and $\neg \Phi$ to create two queries to a SAT solver. Depending on the satisfiability of these two queries, the result of the property checking can be True, False or Maybe. The results generated by the SAT solver are used to generate counterexamples.

With respect to the concerns listed in Table 1.1, making explicit which properties the system must satisfy helps address concerns 2, 5, 28, and 29, while creating support for checking them helps address concerns 10, 19, 22, 34, 35, and 41, since the result of property checking can be used as rationale for making decisions.

1.4.3 Applying transformations

**Challenges** In order to apply transformations in the presence of uncertainty, we must first specify the meaning of transformation for partial models. Our aim is for existing transformations to be directly usable in the presence of uncertainty. We must thus identify how and under what conditions transformations can be systematically adapted to operate on partial models. We must also investigate the effect of our approach on the properties of such transformations.

**Contributions** We address these challenges by using the approach illustrated below and detailed in Chapter 5. Transformations like the FSEA rule in Figure 1.2(d) are typically expected to work on models that do not contain uncertainty. This makes it difficult to transform a partial model: consider for example the state $Seeding$ in $M_{p2p}$, shown in Figure 7.5. Both its incoming transitions are May elements, and the May formula allows creating concretizations with none, one and both of them. Should FSEA be applied to this state? If we were to do so, we could be adding an entry action $share$ to the state $Seeding$ for concretizations without the $CanShare$ alternative!

The intuitive answer is that transforming $M_{p2p}$ should be equivalent to transforming each of its concretizations separately, as shown in Figure 1.5. Therefore $Seeding$ should only be transformed in the subset of concretizations where FSEA is applicable and remain unchanged in the rest.

We show in Figure 1.6 the partial model that has exactly the set of concretizations shown in Figure 1.5. Our aim is to apply transformations such as FSEA directly on $M_{p2p}$ to get this new model, without having to transform each concretization separately.

Applying transformations to partial models helps address concerns 47, and 48 from Table 1.1, since transformation tasks can now be applied to all parts of the model of a software system.

1.4.4 Making decisions

**Challenges** The topic of decision making in the presence of uncertainty is extensively studied in fields of research such as Decision Theory, Design Space Exploration, etc. We scope our approach by focusing on the question of how to leverage partial models to support the resolution of uncertainty. Given the semantics of partial models discussed in Section 1.4.1, we need to first create a formal definition of uncertainty resolution, as well as to define techniques for accomplishing this with partial models.
Figure 1.5: (a-f) Versions of the concretizations in Figure 1.4 after being transformed using the FSEA transformation in Figure 1.2(d).

Subsequently, we must address the issue of the effect of uncertainty resolution to the properties of partial models. Finally, we need to provide a mechanism for tracing the design decisions across the lifecycle of software artifacts.

**Contributions** We address these challenges by using the refinement approach presented in Chapter 6. Specifically, supporting decision deferral means allowing developers to make decisions in a systematic way when the time is right. This can be done either by enforcing properties such as those discussed in
Section 1.4.2, by manually selecting specific desired alternatives as shown in Figure 1.7, or by applying “uncertainty refining transformations”. In all these cases, the objective of decision making is to reduce the number of concretizations of a partial model. In Chapter 7 we introduce Mu-MMINT, a tool that provides support for making decisions, while also generating appropriate traceability information.

This approach helps address concerns 2, 22, 40, 41, 45, and 46 from Table 1.1, since it allows developers to systematically make design decisions while maintaining traceability of the refinement steps and their rationale.

1.5 Organization

The rest of the thesis is organized as follows:

- In Chapter 2, we introduce relevant background literature, concepts and notations.
- In Chapter 3, we present our approach for representing uncertainty using partial models, discussing notational and formal aspects.
Figure 1.7: The elements that reify the “compromise” alternative solution to the decision point “what happens when a download completes” are set to True and highlighted in green. Elements reifying other alternative solutions are set to False and greyed out. Maybe elements that are not part of this decision are left unaltered.

- In Chapter 4, we detail how to perform verification tasks using partial models, and study different techniques for generating user feedback.
- In Chapter 5, we describe a technique for lifting transformations such that they can be applied to partial models.
- In Chapter 6, we study the different ways in which information can be incorporated to partial models in order to resolve uncertainty.
- In Chapter 7, we outline a framework that puts together the tasks described in Chapters 3-6 in a coherent methodology for managing uncertainty.
- In Chapter 8, we illustrate the approach in several worked examples.
- The dissertation concludes in Chapter 9, where we summarize the approach and outline directions for future research.

Various parts of this thesis have been previously peer reviewed and published. Specifically:

- Chapter 1 expands on the content of [Famelis et al., 2011; Famelis, 2012].
- In Chapter 3, the contents of Sections 3.1, 3.3, 3.4, and 3.5 have been published in [Famelis et al., 2012] and expanded in the manuscript [Famelis et al., 2015c], which has undergone two rounds of peer review. The contents of Section 3.2 also expand on previously published material [Famelis, M. and Santosoa, S., 2013].
• The contents of Chapter 4 have been published in [Famelis et al., 2012] and expanded in the manuscript [Famelis et al., 2015c].

• The contents of Chapter 5 have been published in [Famelis et al., 2013].

• The contents of Chapter 6, with the exception of Section 6.1.1, have been published in [Famelis et al., 2012] and expanded in the manuscript [Famelis et al., 2015c].

• The worked-out example in Section 8.1 is has been published in [Famelis et al., 2012].
Chapter 2

Background

In this chapter we introduce definitions and relevant work from the academic literature for concepts that are used in the rest of thesis.

In Section 2.1, we present the mathematical foundation of software modelling and show how to represent software models using propositional logic. In Section 2.2, we introduce the verification problem for software models and show how it can be addressed using the propositional representation of models. In Section 2.3, we describe model transformations and their properties, focusing on transformations based on graph rewriting. Finally, in Section 2.4, we give a high level overview of the Model Driven Architecture, a software engineering methodology that uses models as the primary development artifact.

2.1 Models and Metamodels

2.1.1 Modelling formalisms

The word “model” has many meanings in various fields of human endeavour. In this thesis, we use “model” according to [Kühne, 2005]:

In software engineering “model” has traditionally been referring to an artifact formulated in a modelling language, such as UML, describing a system through the help of various diagram types. In general, such model descriptions are graph-based and typically rendered visually.

Models are used either descriptively, i.e., to abstract an already-existing system, or prescriptively, i.e., to specify a system under development. Therefore, for our purposes, a model is a typed graph that conforms to some metamodel represented by a distinguished type graph. The definitions that follow are based on [Ehrig et al., 2006].

Definition 2.1. A graph is a tuple \( G = (V, E, s, t) \), where \( V \) is a set of nodes, \( E \) is a set of edges, and \( s, t : E \rightarrow V \) are the source and target functions, respectively, that assign each edge a source and target node.

The above definition allows us to construct directed multigraphs that contain parallel edges and self loops. For example, the graph shown in Figure 2.1(b) has two parallel edges between the nodes Idle and

---

1Thomas Kühne describes this definition as “too narrow” and explains a series of nuances that are necessary for a holistic understanding of the notion of a “model” in all software engineering contexts. These nuances, albeit important, are not necessary for this presentation.
Leeching. In the following, we assume that each node and edge has its own identity. For convenience, we express this identity by means of a unique name.

Definition 2.2. A typed graph $G^T$ of type $T$ is a triple $(G, type, T)$ consisting of a graph $G$, a type graph $T$ that represents the metamodel, and a graph homomorphism $type : G \rightarrow T$ that assigns types to the elements of $G$.

For example, the models shown in Figure 2.1 are typed with the type graph shown in Figure 2.2.

In the following, we refer to the typed nodes and typed edges of a model as atoms, or elements.

Definition 2.3. A model $m$ is a typed graph $(G, type, T)$ that satisfies a set $T$ of well-formedness constraints (WFCs) mandated by its metamodel. We represent as $\Phi_T$ the set of all the constraints $\phi_{t_i}$ in $T$ and write $m \models \Phi_T$ to express that $m$ satisfies the WFCs.

We give the set of WFCs $\Phi_{SM}$ for State Machines in Figure 2.2. For example, the first WFC $\phi_{t_1}$ mandates that every Transition must have exactly one source State. For example, in the model $m$ shown in Figure 2.1(a), all transitions have exactly one source node, and therefore, we can write $m \models \phi_{t_1}$.

In order to perform automated reasoning, it is sometimes useful to represent attributed models using typed graphs. Attributes are represented by edges connecting the attributed model element to values in the attribute domain. For example, consider the class Car in Figure 2.3(a) that has an attribute numberOfWheels whose value is 4. The class is shown as a typed graph in 2.3(b). The graph consists of two nodes: a node Car of type Class and a node 4 of type Integer. The attribute is shown as an arrow numberOfWheels from Car to 4.

2.1.2 Propositional encoding of models

In order to reason about uncertainty regarding presence of atoms in a model, it is useful to represent them as propositions. Each proposition expresses whether a particular atom is part of the model or not. In this section, we describe how to encode a model as a set of such propositions.
Chapter 2. Background

Figure 2.3: Representing attributes using typed graphs: (a) Class shown in the concrete UML syntax. (b) The same class shown as a typed graph.

Table 2.1: atomToProposition: mapping model atoms to propositional variables.

<table>
<thead>
<tr>
<th>Atom</th>
<th>Propositional variable</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node n of type q</td>
<td>( n_q )</td>
<td>( \exists n \in V \land \text{type}(n) = q )</td>
</tr>
<tr>
<td>Edge e of type q from x to y</td>
<td>( e_{x \to y , q} )</td>
<td>( \exists e \in E \land \text{type}(e) = q \land s(e) = x \land t(e) = y )</td>
</tr>
</tbody>
</table>

Given a model \( m = \langle G, T, \text{type}, \Phi_T \rangle \) where \( G = \langle V, E, s, t \rangle \), we define a mapping \( \text{atomToProposition} \) which maps atoms of \( m \) to propositions:

- We associate each node atom \( n \) of type \( q \in T \) with a propositional variable \( n_q \) expressing the sentence “the model contains the node \( n \) of type \( q \)”. More formally,

  \[
  n_q \iff \exists n \in V \land \text{type}(n) = q.
  \]

  For example, the state \( \text{Idle} \) in the model in Figure 2.1(a) is mapped to the propositional variable \( \text{Idle} \_\text{State} \).

- We associate each edge \( e \) of type \( q \in T \) with source node \( x \) and target node \( y \) with the proposition \( e_{x \to y \, q} \). It expresses the sentence “the model contains the edge \( e \) of type \( q \) from node \( x \) to node \( y \)”. More formally,

  \[
  e_{x \to y \, q} \iff \exists e \in E \land \text{type}(e) = q \land s(e) = x \land t(e) = y.
  \]

  For example, the transition \( \text{start}() \) in Figure 2.1(a) is mapped to the propositional variable \( \text{start} \_\text{Idle} \_\text{Leeching} \_\text{Transition} \).

The mapping is summarized in Table 2.1.

In the rest of the thesis, we occasionally use shorter aliases in place of the full variable names defined by \( \text{atomToProposition} \) for brevity, provided that the context is unambiguous.

We define the conversion \( \text{graphToPropositionSet} \) of a typed graph to the set of propositions corresponding to its atoms as follows:

\[
\text{graphToPropositionSet}(m) = \{ \text{atomToProposition}(a) \mid a \in V \cup E \}.
\]

For example, for the model in Figure 2.1(a), the encoding \( \text{graphToPropositionSet} \) is:

\[
\{ \text{Idle} \_\text{State}, \text{Leeching} \_\text{State}, \text{Seeding} \_\text{State}, \text{start} \_\text{Idle} \_\text{Leeching} \_\text{Transition}, \text{cancel} \_\text{Leeching} \_\text{Idle} \_\text{Transition}, \text{completed} \_\text{Leeching} \_\text{Seeding} \_\text{Transition}, \text{cancel} \_\text{Seeding} \_\text{Idle} \_\text{Transition}, \text{restart} \_\text{Seeding} \_\text{Leeching} \_\text{Transition} \}
\]
Using the propositional representation allows us to define a simple graph union.

**Definition 2.4.** Let $G^T_1$ and $G^T_2$ be typed graphs with the same metamodel $T$ and let their encodings $\text{graphToPropositionSet}(G^T_1)$ and $\text{graphToPropositionSet}(G^T_2)$ be such that they satisfy the following constraints:

- **C1:** If two atoms have the same name, they must have the same type.
- **C2:** If two edges have the same name, they must have the same source and target nodes.

Then, their graph union is a typed graph $G^T_u$ such that

$$\text{graphToPropositionSet}(G^T_u) = \text{graphToPropositionSet}(G^T_1) \cup \text{graphToPropositionSet}(G^T_2).$$

We write $G^T_u = \text{GraphUnion}\{G^T_1, G^T_2\}$.

**Theorem 2.1.** If the encodings $\text{graphToPropositionSet}(G^T_1)$, $\text{graphToPropositionSet}(G^T_2)$, of two typed graphs $G^T_1$, $G^T_2$ with the same metamodel $T$ satisfy the constraints C1 and C2, then there exists a typed graph $G^T_u = \text{GraphUnion}\{G^T_1, G^T_2\}$.

**Proof.** We define the set $V_u = V_1 \cup V_2$ and the set $E_u = E_1 \cup E_2$. Because the graph union satisfies constraints C1 and C2, we are able to define the functions

$$s_u(e) = \begin{cases} s_1(e) & \text{if } e \in E_1 \\ s_2(e) & \text{if } e \in E_2 \end{cases},$$

$$t_u(e) = \begin{cases} t_1(e) & \text{if } e \in E_1 \\ t_2(e) & \text{if } e \in E_2 \end{cases},$$

$$\text{type}_u(a) = \begin{cases} \text{type}_1(a) & \text{if } a \in V_1 \cup E_1 \\ \text{type}_2(a) & \text{if } a \in V_2 \cup E_2 \end{cases}.$$ 

Therefore, the tuple $\langle G_u, T, \text{type}_u \rangle$, where $G_u = \langle V_u, E_u, s_u, t_u \rangle$, defines a typed graph $G^T_u$. \hfill \Box

For example, the typed graph corresponding to the union of the models in Figure 2.1(a, b) is shown in Figure 2.4(a).

Provided that the constraints C1 and C2 in Definition 2.4 are respected, we can generalize graph union to apply to sets of typed graphs of arbitrary size. Given a set $S$ of typed graphs of type $T$, we denote by $\text{GraphUnion}(S)$ the graph that is constructed by iteratively creating pairwise graph unions of the members of $S$.

Even if the typed graphs used to construct models are well-formed, there is no guarantee that $G^T_u$ is also a well-formed model. For example, consider a simple well-formedness constraint $\phi_t$ mandating acyclicity. The two toy models $m_1, m_2$ in Figures 2.4(b, c) are obviously acyclic. However, the model $\text{GraphUnion}(m_1, m_2)$ shown in Figure 2.4(d) does not satisfy $\phi_t$.

### 2.2 Verification of Software Models

When using models descriptively we can employ automated reasoning techniques to identify defects in the underlying system. When using models prescriptively, automated reasoning is useful to ensure that we do not introduce defects to the system being designed [Dubois et al., 2013]. Both tasks are...
accomplished by expressing properties that we want the system to satisfy. We distinguish two kinds of properties: semantic and syntactic. Semantic properties concern the behaviour of the system, while syntactic – its structure. In this thesis, we focus on syntactic properties (for the rationale behind this scoping, refer to Section 4.1.1).

2.2.1 Encoding of syntactic properties

We consider syntactic properties expressed in first order logic (FOL). (Treatment of other FOL-like languages, such as the Object Constraint Language (OCL) [Object Management Group, 2006b] would be similar.) We assume that the resulting formulas have no free variables, i.e., variables not in scope of some quantifier. For example, in PtPP, the property P2 (“no two transitions have the same source and target”) is expressed in FOL as follows:

\[
\forall t_1, t_2 : \text{Transition} \cdot (\text{Source}(t_1) = \text{Source}(t_2) \land \text{Target}(t_1) = \text{Target}(t_2)) \iff (t_1 = t_2)
\]

In this thesis, we only deal with finite models and thus all quantifiers range over finite universes. For such models, all FOL formulas can be grounded over the finite set of propositional variables in \(\text{graphToPropositionSet}\) (extensive work on FOL grounding is reported in [Jackson, 2006a; Gebser et al., 2011]). We assume that for each FOL-expressed property \(p\), a corresponding ground propositional formula \(\phi_p\) is provided.

For example, the well-formedness constraint \(\phi_t\) in Figure 2.2 can be grounded over the universe of variables in the model in Figure 2.1(a) as follows:

\[
\text{start}\_\text{Idle}\_\text{Leeching}\_\text{Transition} \Rightarrow \text{Idle}\_\text{State} \land \\
\text{cancel}\_\text{Leeching}\_\text{Idle}\_\text{Transition} \Rightarrow \text{Leeching}\_\text{State} \land \\
\text{completed}\_\text{Leeching}\_\text{Idle}\_\text{Transition} \Rightarrow \text{Leeching}\_\text{State} \land \\
\text{cancel}\_\text{Seeding}\_\text{Idle}\_\text{Transition} \Rightarrow \text{Seeding}\_\text{State}
\]

2.2.2 Verification of properties

Automated reasoning aims at verifying whether a model satisfies a given property. The result of property checking is a boolean value: True if the model satisfies the property and False otherwise.

Given the logical encoding of models (described in Section 2.1.2) and properties (described in Section 2.2.1), a simple decision procedure can be defined for property checking:
Definition 2.5. Given a model $m$ and a property $p$, with propositional encodings $\phi_m$ and $\phi_p$ respectively, we check whether the expression $\phi_m \land \phi_p$ is satisfiable using a SAT solver. If $\phi_m \land \phi_p$ is satisfiable, then $m$ satisfies $p$ and we write $m \models p$.

2.3 Model Transformations

2.3.1 Graph rewriting

Model transformations have been described as the “heart and soul of model-driven software development” [Sendall and Kozaczynski, 2003]. Put simply, a model transformation is a model that describes how to change other models. There are numerous approaches and languages for creating and executing model transformations [Czarnecki and Helsen, 2006], however in this thesis we focus on a particular class of transformations that relies on graph rewriting called graph transformations [Ehrig et al., 2006]. Such transformations apply to models such as the model $m_a$ in Figure 2.5(a). They are implemented by executing a set of graphical rules defined as follows:

Definition 2.6 (Transformation rule). A transformation rule $R$ is a tuple $R = \langle \{NAC\}, LHS, RHS \rangle$, where the typed graphs $LHS$ and $RHS$ are respectively called the left-hand and the right-hand sides of the rule, and $\{NAC\}$ represents a (potentially empty) set of typed graphs, called negative application conditions.

To illustrate this we use models and transformations from the example scenario SolvEx, described in more detail Section 5.1. Specifically, we use the example of a transformation that implements the Encapsulate Variable refactoring [Casais, 1994]. A generic method for implementing this refactoring using graph transformations was described by Mens et al. [Mens et al., 2005]. A simplified version of this rule, called $R_{EV}$, is shown in Figure 2.5(e). The left-hand side (LHS) of the rule matches a node $a$ (and its associated edges such as $a_0$) that represents a public attribute. The right-hand side (RHS) makes it private (by deleting the $isPublic$ edge $apt$ from $a$ to True and adding a new $isPublic$ edge $apf$ from $a$ to False). It also creates a public getter operation $ge$ and its associated edges. In addition, the rule has two negative application conditions (NACs), i.e., conditions under which the rule should not be applied. These are: $NAC_1$, specifying the case when the class containing the public attribute is an inner class, and $NAC_2$, specifying the case when the class already has a getter.

The LHS, RHS and NACs of a rule consist of different parts, i.e., sets of model elements which do not necessarily form proper graphs. These parts play different roles during the rule application:

- $C^r$: The set of model elements that are present both in the LHS and the RHS, i.e., remain unaffected by the rule.
- $D^r$: The set of elements in the LHS that are absent in the RHS, i.e., deleted by the rule.
- $A^r$: The set of elements present in the RHS but absent in the LHS, i.e., added by the rule.
- $N^r$: The set of elements present in any NAC, excluding those included in $C^r$.

The parts of the example rule $R_{EV}$ are shown in Figure 2.6. Specifically, $C^r$ is the set $\{a, ao, an, at, c1, ct\}$, $D^r$ is the (unary) set $\{apt\}$, $A^r$ is $\{g, go, gn, gp, gt, apf\}$, and $N^r$ is $\{g, gn, go, gt, c1n, c2\}$.

A rule $R$ is applied to a model $m$ by finding a matching site of its LHS in $m$:

---

2The model $m_a$ is shown in the abstract syntax of the Simplified UML Class Diagram metamodel defined in Figure 2.5(d).
3Nodes that represent values (e.g., boolean True, the string N, etc.) are also considered to be part of $C^r$ but are omitted for brevity.
Definition 2.7 (Matching site). A matching site of a transformation rule $R$ in a model $m$ is a tuple $K = (\overline{N}, C, D)$, where $C$ and $D$ are matches of the parts $C^r$ and $D^r$ of the LHS of $R$ in $m$, and $\overline{N}$ is the set of all matches of NACs in $m$ that are anchored at the matches $C$ and $D$.

For example, a matching site $K_1$ for the rule $R_{EV}$ in the model $m_a$ in Figure 2.5(a) is $(C_1, \overline{N}_1, D_1)$, where $C_1 = \{e, eo, en, et, SolverException, String\}$, $\overline{N}_1 = \{\{Solver, sn\}\}$, and $D_1 = \{ept\}$.

In the above definition, $\overline{N}$ denotes the set of all matches within $m$ of the NACs of $R$. Given the match of $C^r$ and $D^r$. If the same NAC can match multiple ways, then all of them are included in $\overline{N}$ as separate matches. For example, if the model in Figure 2.5(a) had another class Solver2 that also nested SolverException via an edge $sn2$, then $\overline{N}$ would contain two matches for NAC$_1$: $\overline{N} = \{\{Solver, sn\}, \{Solver2, sn2\}\}$. The set of matching sites define the places in the model $m$ where the rule can potentially be applied.

Definition 2.8 (Applicability condition). Given a transformation rule $R$, a model $m$, and matching site $K = (\overline{N}, C, D)$, the rule $R$ is applicable at $K$ iff $\overline{N}$ is empty$^4$.

$^4$The theory of graph transformation requires some additional formal preconditions, most notably the gluing condition [Ehrig et al., 2006]. In the following we always assume that these are satisfied.
The above definition ensures that the rule can only be applied at a given site if no NAC matches. For $R_{EV}$, the matching site $K_1$ in $m_a$ does not satisfy the applicability condition as $\mathbb{N}_1 \neq \emptyset$. On the other hand, the model $m_b$ in Figure 2.5(b) contains a matching site $K_2 = \langle \emptyset, C_1, D_1 \rangle$, which does satisfy this condition. Then, the rule can be applied:

**Definition 2.9 (Rule application).** Given a transformation rule $R$, a model $m$, and a matching site $K$ in $m$ for which the rule applicability condition is satisfied, rule $R$ is applied, producing a model $m'$, by removing $D$ from $m$ and adding $A$, where $A$ is a match of the part $A^r$ of $R$ in $m$. Rule application is denoted as $m \xrightarrow{R} m'$.

Applying $R_{EV}$ to $m_b$ at $K_2$ thus requires the deletion of the element $ept$ because it is contained in $D$, and the addition of new elements according to $A^r$. The resulting model $m_c$ is shown in Figure 2.5(c), where $A$ is the set $\{ge, go, gn, gp, gt, ept\}$.

We refer to rules such as the ones described above as “classical” to differentiate them from their “lifted” counterparts which are discussed in Chapter 5.

### 2.3.2 Properties of transformations

A **graph rewriting system** (GRS) is a set of graph rewriting rules plus an input model. A GRS defines a particular setting in which graph-rewriting-based model transformations take place. Graph rewriting systems are characterized by two important properties, namely, **confluence** and **termination**. The definitions in this section are based on [Nupponen, 2005; Ehrig et al., 2006].

In the following, given a set of graph rewriting rules $S$, we denote the application of any rule in $S$ with the symbol $\xrightarrow{S}$. Repeated application of rules from $S$ is denoted with the symbol $\xrightarrow{S^*}$.

Consider the toy example model $C_1$, shown in Figure 2.7(a), which contains three **Classes**: Pop, Soda, and Coke. The rule $F_1$, shown in Figure 2.7(e) transforms every **Class** $x$ to a new **Table**, deleting $x$. Applying the rule $F_1$, shown in Figure 2.7(e) to $C_1$, results in the model $C_2$, shown in Figure 2.7(b), which only contains **Table** elements. Since $C_2$ no longer contains any **Classes**, the rule $F_1$ can no longer be applied to it. In contrast, consider the rule $F_2$, shown in Figure 2.7(f), which transforms every **Class** $x$ to a new **Table**, without deleting $x$. Applying $F_2$ to $C_1$ results in the model $C_3$, shown in Figure 2.7(c), which contains the new **Tables**, as well as the original classes. $F_2$ can therefore be applied again to $C_3$, resulting in the model $C_4$, shown in Figure 2.7(d). $C_4$ contains all elements of $C_3$, as well as new copies.
of all three Tables. It is thus possible to repeatedly apply F2, adding new copies of Tables ad infinitum. The GRS \(\{\{F1\},C1\}\) is called terminating, whereas the GRS \(\{\{F2\},C1\}\) is called non-terminating.

We define termination for a GRS with as follows:

**Definition 2.10** (Termination). Given a set \(S\) of graph rewriting rules and an input model \(m_0\), the GRS \(G = \langle S,m_0 \rangle\) is terminating iff there is no infinite sequence of rule applications \(m_0 \xrightarrow{S} m_1 \xrightarrow{S} \ldots\).

Consider now the rule F3, shown in Figure 2.7(g), which merges two Classes \(u, v\) to a single Table \(uv\). There are three matching sites of F3 in C1: \(k_1=(\text{Pop}, \text{Soda})\), \(k_2=(\text{Pop}, \text{Coke})\), and \(k_3=(\text{Soda}, \text{Coke})\). Depending on which matching site in C2 we apply F3 first, we get different models. For example, applying it to \(k_1\) results in the model C5, shown in Figure 2.7(h), which contains the Table \(\text{PopSoda}\) and the Class \(\text{Coke}\); however, applying it to \(k_2\) results in the model C6, shown in Figure 2.7(i), which contains the Table \(\text{PopCoke}\) and the Class \(\text{Soda}\). For GRSs with multiple rules, the choice of which rule to apply is an additional source of non-determinism. In contrast, it does not matter in what order we find matching sites of F1 in C1: the result is always the model C2. The GRS \(\{\{F1\},C1\}\) is called confluent, whereas the GRS \(\{\{F3\},C1\}\) is called non-confluent.

We define confluence for a GRS as follows:

**Definition 2.11** (Confluence). Given a set \(S\) of graph rewriting rules and an input model \(m_0\), the GRS \(G = \langle S,m_0 \rangle\) is confluent iff for all models \(m_1, m_2\) such that \(m_0 \xrightarrow{S} m_1 \xrightarrow{S} \ldots\) and \(m_0 \xrightarrow{S} m_2 \xrightarrow{S} \ldots\) there is a model \(m_h\) such that \(m_1 \xrightarrow{S} m_h\) and \(m_2 \xrightarrow{S} m_h\).

### 2.4 Model-Driven Architecture

The Model Driven Architecture (MDA) [Object Management Group, 2014] is a software development methodology introduced by the Object Management Group (OMG) in 2001. Relying on a series of OMG standards, such as the Meta Object Facility (MOF) [Object Management Group, 2006a], the Unified Modeling Language (UML) [Object Management Group, 2015a], the XML Metadata Interchange
(XMI) [Object Management Group, 2015b], and others, the MDA is OMG’s canonical realization of Model Driven Engineering (MDE).

In MDA, models, metamodels and model transformations are first class development artifacts. Software applications are created by applying transformations on models expressed at a high level of abstraction to derive models at lower levels of abstraction.

MDA promotes separation of concerns by adopting a design pattern whereby developers separate the functionality of a system from the resources on which it depends in order to deliver it. A coherent set of such resources is called a “platform”. Concerns are separated in the sense that developers can create “Platform Independent Models” (PIMs), that abstract away the details of the platform. Then, using transformations that take PIMs and platform models as inputs, they can derive “Platform Specific Models” (PSMs) that include the additional necessary details at a lower level of abstraction.

Levels of abstraction are captured as “architectural layers”. The PIM/PSM pattern reflects the crossing of the boundary between consecutive layers. Three such layers are defined:

1. the layer of business or domain models,
2. the layer of logical system models, and
3. the layer of implementation models.

Transformations are used both to cross architectural layer boundaries, as well as to derive different models within the same layer, such as models for analysis, documentation, etc.

In the first layer, developers model the relevant parts of real world, without worrying about how its elements will be represented in the software system. Since this is the top layer, its models are known as “Computation Independent Models” (CIMs).

Models in the second layer capture the high level architecture of the software system: its parts and the interactions between them. Logical system models are PSMs with respect to CIMs. Specifically, they are derived from the business layer using transformations that take as input CIMs and a set of design decisions (which constitute the “platform” for this layer).

The third layer contains models that describe particular software artifacts, with enough information such that they can functionally achieve the goals of the application. Implementation models are PSMs with respect to logical system models. Specifically, they are derived from the second layer using transformations that take as input the logical system models, and a description of the implementation platform. Typical examples of such transformations are code generators.

Consider the example of an application for managing hotel bookings. At the business level, developers create CIMs that describe domain concepts such as rooms, dates, customers, etc. The CIMs are platform independent with respect to logical system models in the next layer, since they do not contain information such as client/server architectures, abstract data types, etc. The logical system models themselves are platform independent compared to implementation models, that describe the Java source code realizing the system.

### 2.5 Summary

In this chapter, we have introduced definitions and concepts that form the foundation for the techniques presented in the rest of the thesis. We provided definitions for models and metamodels, which are used
throughout the thesis. The propositional encoding of models in particular is used in Chapters 4 and 6. We also introduced the basics of model verification, which are used in Chapter 4. The theory of graph-rewriting based model transformations is the basis for the technique presented in Chapter 5. Finally, the description of Model Driven Architecture is used in Chapter 7.
Chapter 3

Representing Uncertainty

In this chapter, we describe how design-time uncertainty can be represented using partial models. We introduce the formal semantics of partial models and describe equivalent reduced forms which can represent the same set of concretizations but are best suited for different developer activities (human modelling and comprehension; automated reasoning). We describe in detail the notation of the partial modelling language, along with the rationale behind its design. We define a process of partialization through which existing modelling languages can be adapted to support partial modelling and evaluate the notation using an established theory for designing graphical modelling languages, called “Physics of Notations”. Finally, we give an algorithm for constructing partial models from a set of concrete models. This way, a developer who is faced with uncertainty about which alternative model she should use can create a partial model that exactly and compactly encodes the set of available solutions.

We use the example PtPP scenario introduced in Chapter 1, whereby a team of developers is designing a peer-to-peer system and is uncertain about certain decisions, as shown in Figure 1.2(a). The developers explicate their uncertainty using the partial model shown in Figure 1.3, which encodes the six possible resolutions of uncertainty shown in Figure 1.4.

The rest of this chapter is organized as follows: in Section 3.1 we define the formal semantics of partial models. In Section 3.2 we give a complete description of the partial modelling notation, describe how to adapt existing modelling languages to support it, and evaluate it using the “Physics of Notations” theory. In Section 3.3 we introduce two reduced forms, i.e., equivalent representations of partial models. In Section 3.4 we give the algorithm for constructing a partial model from a set of non-partial models. We discuss related work in Section 3.5, and summarize and conclude in Section 3.6. Proofs of theorems are found in Appendix A.1.

The contents of Sections 3.1, 3.3, 3.4, and 3.5 have been published in [Famelis et al., 2012] and expanded in the manuscript [Famelis et al., 2015c], currently under review. The contents of Section 3.2 also expand on previously published material [Famelis, M. and Santos, S., 2013].

3.1 Semantics

In this section, we formally define partial models and their associated operations. Semantically, a partial model represents a set of classical (i.e., non-partial) models. The particular type of partiality we consider in this paper is the one that allows a modeler to express uncertainty as to whether particular model
atoms should be present in the model. The model is accompanied by a propositional formula, called the
May formula, which explicates allowable combinations of such atoms.

**Definition 3.1.** A Partial Model is a tuple \( \langle G^T, \text{ann}, \phi \rangle \), where \( G^T = \langle \langle V, E, s, t \rangle, \text{type}, T \rangle \) is a typed
graph, \( \text{ann} : V \cup E \rightarrow \{ \text{True}, \text{Maybe} \} \) is a function for annotating atoms in \( G \), and \( \phi \) is a proposi-
tional May formula over the set of propositional variables of \( \text{Maybe} \)-annotated atoms, i.e., the set
\{atomToProposition(a)|a \in V \cup E \land \text{ann}(a) = \text{Maybe}\}.

In the above definition, an annotation \text{True} means that the atom must be present in the model,
whereas \text{Maybe} indicates uncertainty about whether the atom should be present. In other words, a
partial model consists of a typed graph whose atoms are annotated with \text{True} or \text{Maybe}, and a May
formula that describes the allowed configurations of its atoms.

Model \( M_{p2p} \) in Figure 1.3 is an example of a partial model. The atoms decorated with the seven-
pointed star icons, such as the state \text{Finishing} and the transition label \text{completed()}, are \text{Maybe} elements.
Atoms annotated with \text{True}, e.g., the state \text{Idle} and the transition action \text{start}, do not have special
decorations. \( M_{p2p} \) is accompanied by the May formula \( \Phi_{p2p} \) shown below it in the figure. We have used
shorthand aliases to stand in for the full names of the propositional variables that correspond to \text{Maybe}
atoms. For example, \( Fs \) is an alias for the variable \text{Finishing \_ \_ State}.

Given a partial model \( M \), we can obtain a classical model \( m \) by making a choice to keep or discard
each \text{Maybe}-annotated atom while satisfying the May formula constraint:

**Definition 3.2.** A concretization \( m \) of a partial model \( M = \langle G, \text{ann}, \phi \rangle \) is a well-formed classical model
such that

- \( \forall a \in m \Rightarrow a \in G; \)
- \( \phi' = \text{True} \), where \( \phi' \) is obtained by substituting all variables in \( \phi \) using the rule:
  - \( a \in m : \phi[\text{True}/\text{atomToProposition(a)}] \), i.e., replace a’s encoding by \text{True}; and
  - \( a \notin m : \phi[\text{False}/\text{atomToProposition(a)}] \), i.e., replace a’s encoding by \text{False}.

For example, the concretizations of \( M_{p2p} \) are the models shown in Figures 1.4(a-f).

We define the semantics of partial models as follows:

**Definition 3.3.** Given a partial model \( M \), we define its semantics as the set of all concretizations that
can be derived from it using Definition 3.2. We denote this set as \( C(M) \). If \( C(M) \) is empty, we say that
\( M \) is inconsistent.

In the PtPP example, \( C(M_{p2p}) \) consists of the models shown in Figures 1.4(a-f).

In what follows, we only assume consistent partial models.

An important characteristic of partial models is that the size of the set of concretizations reflects
the modeler’s degree of uncertainty. A classical model has no uncertainty and uniquely corresponds
to a partial model with a single concretization (the uniqueness follows from Theorem 3.5, given in
Section 3.4).

**Definition 3.4.** A singleton partial model is a partial model \( M \) whose set of concretizations \( C(M) \)
contains exactly one concretization.
Using partial model semantics, we can apply a formal approach to changing the degree of uncertainty in a model, through a process of refinement:

**Definition 3.5.** Given two partial models $M_1$ and $M_2$, we say that $M_2$ refines $M_1$ (or that $M_1$ is more abstract than $M_2$), denoted $M_2 \preceq M_1$, iff $C(M_2) \subseteq C(M_1)$, and $M_2$ is consistent.

For example, the model $M_{p2}^\neg P$ in Figure 3.4(a) is more refined than $M_{p2}$ in Figure 1.3. In particular, $C(M_{p2})$ consists of the models in Figure 1.4(a)-(f), whereas $C(M_{p2}^\neg P)$ consists just of those in Figure 1.4(c,d). Thus, $M_{p2}^\neg P$ has less uncertainty. We discuss the pragmatics of refinement in Chapter 6.

Given a partial model $M$, Definition 3.5 allows us to construe its concretizations as the ultimate result of a series of refinements.

**Lemma 3.1.** Given a partial model $M$, each concretization $m_i \in C(M)$ corresponds uniquely to a singleton partial model $M_i$ such that $C(M_i) = \{m_i\}$. Obviously, $M_i \preceq M$.

In the following, we don’t make the distinction between a concretization and its corresponding singleton partial model and abbreviate this to $m_i \preceq M$.

Lemma 3.1 allows us to provide an alternative definition of partial model semantics in terms of the refinement relationship.

**Definition 3.6.** The semantics of a partial model $M$ is the set of all non-partial models $m_i$ such that $m_i \preceq M$.

### 3.2 Notation

We introduced the partial model notation in Section 1.4.1. We give the full description below:

1. A partial model consists of two parts: a diagram and a May formula.

2. Elements of the diagram that are annotated with **Maybe** are decorated with a special icon (a black seven-pointed star) which contains a propositional variable in white font used to represent the proposition “the element is part of the model”.

3. The star decoration is placed on top or near its corresponding **Maybe** element, depending on its concrete syntax. Specifically, it is placed:
   - at the border of node elements,
   - on top of vertex elements, and
   - next to elements that are labels.

4. Elements that are annotated with **True** have no special decorations.

5. The May formula is placed either below or next to the diagram.

6. The May formula is expressed in the language of propositional logic, that uses propositional variables, logical connectives (And ($\land$), Or ($\lor$), Not ($\neg$), Biconditional ($\leftrightarrow$), etc.) and parentheses.
This notation is used throughout this document and is most convenient for partial models that are typeset using a modeling tool. We have in the past used variations of this notation, such as indicating Maybe elements with dashed lines and borders, and using the textual annotation “(M)”.

For hand-drawn partial models created on paper or a board, the seven-pointed star icon may be cumbersome, especially because of the coloring requirement (white font on a black shape). In such cases of informal modelling, developers tend to use ad-hoc graphical notations that best fit their sketching needs. We have found the textual annotation “(M)” to be practical.

3.2.1 Design rationale

The partial model notation was influenced by preexisting partial behavioural modelling notations [Larsen and Thomsen, 1988], where the focus is on creating a notation that is appropriate for automated reasoning. Since working in the presence of uncertainty is a challenging mental task of its own, we conducted an empirical user study [Famelis, M. and Santosa, S., 2013] to further investigate what is an appropriate notation that developers can use effectively for modelling.

The main focus of the study was to compare two alternative notations for expressing a generalized uncertainty theory called MAVO [Salay et al., 2012b]. Its specific research questions and findings are therefore outside of the scope of this thesis. However, as a warm-up exercise to the main questionnaire, participants were asked to perform a free-form modelling task where they were asked to come up with their own concrete syntax for expressing uncertainty. The serendipitous findings of this exercise significantly influenced the development of the partial modelling notation.

Specifically, participants were presented with a model in a well-known modelling language, namely, an Entity-Relation (E-R) diagram [Chen, 1976], and were asked to express two uncertainty scenarios. Two examples of user-created notations collected during the study are shown in Figure 3.1. The participants'
sketches can be seen in the lower part of the figure.

In the left sketch, we can see that the participant has used dashed lines and question marks to indicate uncertainty in the model. The participant has tried to indicate points of uncertainty using grouping: a cluster of BlogEntry elements have been enclosed in a dashed lasso line, annotated with a question mark. A second cluster of stacked elements named VisitorStatistics is dependent on the first cluster. Additionally, a pair of dashed links to the Author element seem to be alternatives to each other, and the choice seems to also be dependent on the existence of the BlogEntry cluster.

In the right sketch, the participant has avoided changing the E-R diagram notation, with the exception of the VisitorStatistics elements, that are drawn using dashed lines and ellipses. The participant has tried to indicate the uncertainty in the model by annotating the different model elements with numbered choices (indicated by the letter C) and numbered alternatives (indicated by “Alt”). For example, the entity Author appears twice: once connected to Blog (annotated as alternative #1 of choice #2) and once connected to BlogEntry (annotated as alternative #2 of choice #2).

We note two themes: (a) the participants tried to express uncertainty by extending the existing concrete syntax, and (b) they avoided using mathematical or logical formulas, and instead came up with ad-hoc ways to enumerate alternatives on the model and their dependencies. These themes are consistent with research in the field of feature modelling, where it has been shown that many useful propositional formulas can be efficiently created graphically by humans [Czarnecki and Wasowski, 2007].

We thus chose to create a partial modelling notation that only minimally alters the concrete syntax of the underlying language. We chose to use the seven-pointed star icon in order to avoid as much as possible confusion with potential existing graphical symbols of underlying languages (more on this in Section 3.2.2) because it is a symbol rarely used in graphical modelling languages (to our best knowledge, there is no graphical language for modelling software artifacts that uses it).

We also developed a purely graphical language for expressing logical dependencies between Maybe elements without using the language of propositional formulas. In this, we were guided by the principles described by D. Moody’s “Physics of Notations” theory [Moody, 2009], with the aim to maximize the effectiveness of the notation for reading and writing tasks. Even though we have evidence [Famelis, M. and Santosa, S., 2013] that this language is more usable for expressing and comprehending partial models, it has limited (logical) expressiveness and is difficult to use when manipulating partial models (e.g., when applying transformations, as described in Chapter 5). We therefore do not use it in the examples in this dissertation. Nonetheless, it was very influential in the development of the graphical tool Mu-MMINT, which is described in Chapter 7.

### 3.2.2 Base language metamodel adaptation

Our approach for representing uncertainty is modelling-language independent, in the sense that it allows uncertainty to be captured in any model that is expressed in a modelling language defined using a metamodel [Atkinson and Kuhne, 2003; Object Management Group, 2006a]. In effect, the language of partial models is an annotation language that can be superimposed on any graphical modelling language. In this section, we describe the process by which the metamodel of any existing modelling language can be adapted for partial modelling, such that it can be used in modelling tools.

In a given modelling scenario, we call the modelling language of the models without uncertainty the "base language". In the PtPP example, the developers are working with UML state machines, thus the base language is the one defined by the metamodel in Figure 1.2(c).
Given a base modelling language, we derive its corresponding partial language through a process which we call metamodel partialization.

**Definition 3.7.** Given a base language metamodel $M_B$, its corresponding partial metamodel $M_P$, called a partialized metamodel, is a metamodel that contains the same meta-elements as $M_B$, with the following adaptations:

- Every meta-element has an additional boolean meta-attribute $\text{isMaybe}$.
- $M_P$ contains an additional singleton meta-class $\text{MayFormula}$, that has a single attribute $\text{formula}$.
- The multiplicity constraints of every meta-association are unbounded (typically denoted by “*”).

The boolean meta-attribute $\text{isMaybe}$ is used to represent the annotation function $\text{ann}$ in Definition 3.1. Specifically, for each instance element $e$ for which $\text{ann}(e) = \text{Maybe}$, the meta-attribute is set to True; otherwise, it is set to False.

The meta-class $\text{MayFormula}$ is used to store the partial model’s propositional May formula in its attribute $\text{formula}$. It is a singleton class (i.e., it has exactly one instance) since a partial model can only have one May formula. We have intentionally not prescribed a specific data type for this attribute. A String is sufficient; however, toolsmiths may choose to use more sophisticated data structures such as Binary Decision Diagrams (BDDs) [Akers, 1978] or languages specifically tailored for representing logic, such as SMT-LIB [Barrett et al., 2010].

To motivate the need to relax multiplicity constraints of meta-associations, consider for example the simple metamodel $\mathcal{B}$ shown in Figure 3.2(a) that defines a toy language for describing arrangements of boxes. In this language, a Box can contain up to five other Boxes and can be contained in up to one other Box. Consider also the partial model shown in Figure 3.2(c), that encodes a scenario where we are uncertain whether the Box $b$ should be contained in the Box $a$ or the Box $c$. The uncertainty is
expressed by drawing two Maybe annotated containment edges originating from \( b \) and ending at \( a \) and \( c \), respectively. The May formula (an instance of the singleton meta-class \( \text{MayFormula} \) of \( B_p \)) enforces that only one of the two edges can be included in concretizations. This partial model clearly violates the multiplicity constraints of the metamodel \( B \): the Box \( b \) has two containment associations, whereas it is allowed to have at most one. On the other hand, its concretizations are well formed \( B \) models. Relaxing the multiplicity constraints of the base language metamodel, as shown in the partial metamodel \( B_p \) in Figure 3.2(b), allows expressing such scenarios without breaking metamodel conformance rules.

Not all instances of a partialized metamodel are consistent partial models, i.e., it is possible to instantiate partial models that do not have any well-formed concretizations. Partial model editors should therefore support consistency checking. This can be accomplished by combining the May formula with the base language well-formedness constraints and checking the combination using a SAT solver. In the following, we only consider consistent partial models.

It is also possible to instantiate consistent partial models that are not expressible in the concrete syntax of the base language. This is especially true because of the unbounded multiplicities of the partialized metamodel. For example, the result of metamodel partialization to the simplified UML State
Machine metamodel used in PtPP, in Figure 3.3(b). We have copied in Figure 3.3(a) the metamodel from Figure 1.2(c). The partial model in Figure 1.3 conforms to this partial metamodel. Consider the model fragment of \( M_{p2p} \) shown in Figure 3.3(c), consisting of the state \textit{Leeching} and the two \textit{Maybe}-annotated transitions on \textit{completed} towards \textit{Idle} and \textit{Seeding}. Because the partialized metamodel allows unbounded multiplicities, it is possible to model the same scenario using the arrangement of \textit{Maybe} elements shown in Figure 3.3(d). Specifically, instead of using two different \textit{Maybe}-annotated transitions, we have modelled the same scenario with a single \textit{Transition} \( tc \), that has two \textit{Maybe}-annotated \textit{tgt} arrows to the two target states. This way of modelling the PtPP scenario might be more intuitive to the modeller; however, it is not expressible in the concrete UML State Machine syntax where \textit{Transitions} are drawn as arrows themselves. This example demonstrates that the partial model annotation language is versatile and can be used in whatever way is most intuitive to developers.

Definition 3.4 and Lemma 3.1 allow us to establish the correspondence between instances of the base and partialized metamodels:

\textbf{Theorem 3.1.} Every well formed base language instance model can be uniquely transformed into a singleton partial model, instance of the partialized metamodel, and vice versa.

Specifically, a base language instance can be transformed as an instance of the partialized metamodel by adding a meta-attribute \textit{isMaybe} to each element and setting it to \textit{True}, and by including an instance of the class \textit{MayFormula}, with the \textit{formula} attribute set to \textit{True}. Conversely, every singleton partial model, i.e., a (consistent) partial model that has exactly one concretization can be transformed as an instance of the base language by dropping the \textit{isMaybe} meta-attribute from each element and deleting the instance of the \textit{MayFormula} class. Since the partial model is consistent, its single concretization is (by definition) well-formed with respect to the base language metamodel.

Using the partialization technique, existing modelling tools that support metamodelling can be adapted to support partial models. We note that the objective of metamodel partialization is to aid the creation of modelling software, not to set the mathematical foundation of partial models. Therefore, in the rest of the thesis, we use the formalization of partial models as given in Definition 3.1.

\subsection*{3.2.3 Assessment}

In this section, we assess the partial model annotation language using the theory of “Physics of Notations” [Moody, 2009], a design theory focused on creating cognitively effective visual notations. The theory provides a comprehensive framework for evaluating the perceptual properties of graphical languages based on a set of well-defined design principles.

The results of our assessment are summarized below. For each principle, we assign a rating, ranging from + + (“very good”) to - - (“very bad”). We additionally use +/- to mean “it depends”.

### Chapter 3. Representing Uncertainty

<table>
<thead>
<tr>
<th>Design principle</th>
<th>Assessment Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semiotic Clarity</strong>&lt;br&gt;(1:1 symbol-concept correspondence)</td>
<td>+ + Maybe annotations have one-to-one correspondence to partial model semantics. Star symbol is rare in base languages.</td>
</tr>
<tr>
<td><strong>Perceptual Discriminability</strong>&lt;br&gt;(Distinguishable symbols)</td>
<td>+ + Black star icon clearly distinguishes Maybe elements giving them perceptual popout.</td>
</tr>
<tr>
<td><strong>Semantic Transparency</strong>&lt;br&gt;(Representation suggests meaning)</td>
<td>- - Star annotation not easily associated with the concept of uncertainty. Relationships across Maybe elements not visible; relies only on May formula.</td>
</tr>
<tr>
<td><strong>Complexity Management</strong>&lt;br&gt;(Explicit mechanisms handling complexity)</td>
<td>- - New annotation element introduced for each element with uncertainty. No mechanism for chunking information to manage additional complexity.</td>
</tr>
<tr>
<td><strong>Cognitive Integration</strong>&lt;br&gt;(Global view from different diagrams)</td>
<td>- No specific mechanisms, but the May formula contextualizes Maybe elements to overall uncertainty. No mechanism provided for associating them with design decisions at a higher level of abstraction.</td>
</tr>
<tr>
<td><strong>Visual Expressiveness</strong>&lt;br&gt;(Symbols denoted with multiple visual properties)</td>
<td>+ Black, star-shaped icons coupled with textual encoding: measures to three-degrees of visual expressiveness.</td>
</tr>
<tr>
<td><strong>Dual Coding</strong>&lt;br&gt;(Text complements graphics)</td>
<td>- No dual coding; May formula is separated with no visual connection with referent elements: principle of spatial contiguity violated.</td>
</tr>
<tr>
<td><strong>Graphic Economy</strong>&lt;br&gt;(Use of manageable number of symbols)</td>
<td>+ Use of a single graphic symbol.</td>
</tr>
<tr>
<td><strong>Cognitive Fit</strong>&lt;br&gt;(Suitable for audience and medium)</td>
<td>+/- Appropriate for representational mediums of pen-and-paper and tools. Requires skill in propositional logic for May formula.</td>
</tr>
</tbody>
</table>

Overall, the partial model notation is particularly effective in terms of Perceptual Discriminability (the star icon “pops out”) and has a moderate degree of Visual Expressiveness. Since the annotation language is fairly simple (a single icon, along with propositional logic) and the star icon is a rare visual construct in base modelling languages, the notation is clear (Semiotic Clarity) and economical (Graphic Economy). This was by design: since partial models are an annotation language, we cannot make any assumptions about the concrete syntax of the base language. It is thus important to use a symbol that is uncommon in modelling languages so that the combination is not ambiguous. Other notations with which we have experimented, such as using dashed lines are not as effective in this respect, since dashed lines are sometimes used in the concrete syntax of base modelling languages (e.g., to represent constructs such as lifelines and optional blocks in UML Sequence Diagrams).

Additionally, the notation assumes familiarity with propositional logic and modelling tools, and thus its Cognitive Fit is dependent on the skills of the developers using it. Since boolean expressions are a common feature of general purpose modelling languages, this is not an unreasonable expectation.

The Semantic Transparency of the notation suffers because the star icon does not intuitively invoke the concept of uncertainty. In our pilot user study [Famelis, M. and Santosa, S., 2013], participants were asked to express uncertainty in a modelling scenario using whatever symbols were intuitive to them. We found that they often used metaphors such as ellipses, dashed lines and question marks. As discussed above, we chose not to use dashed lines in order to avoid semiotic confusion with constructs of the base language. We chose not to use ellipses because of the low visual pop out, which hurts Perceptual Discriminability. The book *Elements of UML Style* [Ambler, 2005] contains the following recommendation regarding question marks:
“While you are modelling, you may discover that you do not have complete information. [...] You should always try to track down a sufficient answer, but if you cannot do so immediately, then make a good guess and indicate your uncertainty. [A] common way to do so [is using] question marks.”

Question marks are therefore associated with specific semantics regarding the representation of uncertainty: indicating *provisional decisions*. Overloading the symbol would thus risk creating confusion and seriously harm Semiotic Clarity, especially since there are opposing underlying assumptions about how to model uncertainty (the goal of partial modelling is to *avoid* making provisional decisions). Ultimately, we chose to prioritize Semiotic Clarity and Perceptual Discriminability over Semantic Transparency, reasoning that the small learning curve (becoming familiarized with the meaning of one symbol) is not an unreasonable trade-off for the elimination of ambiguity.

The simplicity of the language mitigates the need for Dual Coding. On the other hand, this simplicity means that the notation lacks a good system for Complexity Management, which can be a problem as more and more uncertainty is encoded in a model. Similarly, since the focus is on expressing uncertainty in a single modelling artifact, there are no explicit Cognitive Integration mechanisms, especially in terms of providing an explicit mechanism for mapping *Maybe* elements to design decisions (if a mechanism for capturing them exists). In [Famelis, M. and Santosa, S., 2013] we experimented with grouping *Maybe* elements based on the point of uncertainty with which they are associated. We further elaborated this idea in [Famelis et al., 2015a], by further organizing *Maybe* elements based on which specific alternative solution to a point of uncertainty they reify, as well as explicitly keeping track of points of uncertainty and alternative solutions in a separate modelling artifact called “Uncertainty Tree”. Such organizational mechanisms significantly enhance Complexity Management and Cognitive Integration, and are effective for modelling, reasoning and refining partial models. However, they are not well suited to changes caused to a partial model due to transformations (discussed in Chapter 5) because transformations affect partial models at a lower level of granularity, i.e., per element, thus negating the benefits of element groupings. In this dissertation we aim to present all partial model operators, listed in Section 7.2. Therefore we use the notation that is well suited for all of them.

### 3.3 Reduced Forms

There are multiple ways of representing a given set of classical models using partial models. For example, consider the set of models $S_{\neg P2}$ consisting solely of the two models in Figures 1.4(c, d). These models are the concretizations of $M_{P2p}$ that violate the property $P2$: “no two transitions have the same source and target”. There exist multiple partial models $M_i$ for which $C(M_i) = S_{\neg P2}$. Figure 3.4 shows three such examples. The model in Figure 3.4(a) has only two *Maybe*-annotated atoms and a May formula $Ca \iff Ct$. The model in Figure 3.4(b) has all of its atoms annotated with *Maybe* but a more complex May formula. Both partial models have the exact same set of concretizations $S_{\neg P2}$ but have different combinations of *Maybe* annotations and May formulas. Figure 3.4(c) shows a model that contains both *Maybe*-annotated and *True*-annotated elements. However, its May formula excludes two of the *Maybe*-annotated elements (the transition from *Leeching* to *Seeding* and its corresponding action *completed*) from all concretizations.

**Definition 3.8.** Two partial models $M_1$, $M_2$ are equivalent, denoted $M_1 \sim M_2$, iff they have the same set of concretizations, i.e., $C(M_1) = C(M_2)$. It holds that $M_1 \sim M_2$ iff $M_1 \preceq M_2$ and $M_2 \preceq M_1$. 
Figure 3.4: Some equivalent partial models that encode the set $S_{\neg P^2}$, consisting of the two concretizations in Figures 1.4(c, d): (a) in GRF (b) in PRF (c) containing two superfluous Maybe-annotated atoms ($B_t$ and $B_a$).

To help represent models, we define two special forms: *Graphical Reduced Form* (GRF) and *Propositional Reduced Form* (PRF). Intuitively, a model in GRF represents most information in the graph, whereas in PRF, it represents most of the information in the formula. For example, Figure 3.4(a,b) shows the GRF and the PRF versions, respectively, of the model $M_{\neg P^2}$ in Figure 3.4(c). GRF is the form used...
when a human works with a partial model while PRF is the form used for automated reasoning. We use both forms in defining algorithms in Chapter 4.

We refer to constructing an equivalent GRF (PRF) of a partial model as “putting it to GRF (PRF)”.

In the following, the symbol “⊨” denotes logical entailment.

All algorithms construct models in their abstract syntax. We assume that putting models in their concrete syntax is done automatically by modeling tools.

### 3.3.1 Graphical Reduced Form (GRF)

**Definition 3.9.** A GRF of a partial model $M = \langle G, \text{ann}, \phi \rangle$ is a partial model $M^{GRF} = \langle G^{GRF}, \text{ann}^{GRF}, \phi^{GRF} \rangle$, such that $M \sim M^{GRF}$ and there does not exist $M' \sim M$, such that $M'$ has fewer Maybe-annotated atoms than $M^{GRF}$.

A partial model can have many equivalent GRF models. All GRF equivalents have the same number of Maybe-annotated atoms.

Algorithm 1 describes how to put an arbitrary partial model into its GRF form.

**ALGORITHM 1: GRF Conversion.**

**Input:** A partial model $M = \langle G, \text{ann}, \phi \rangle$

**Output:** A GRF equivalent $M^{GRF} = \langle G^{GRF}, \text{ann}^{GRF}, \phi^{GRF} \rangle$ of $M$

1. $G^{GRF} := G$;
2. $\phi^{GRF} := \phi$;
3. $B = \text{backbone}(\phi)$;
4. foreach atom $a \in G$ do
5.   if ann($a$) = Maybe then
6.     if atomToProposition($a$) $\in B$ then
7.       ann$^{GRF}(a) := \text{True}$;
8.       substitute $\phi^{GRF}[\text{True}/\text{atomToProposition}(a)]$;
9.     else if ($\neg\text{atomToProposition}(a)) \in B$ then
10.        remove $a$ from $G^{GRF}$;
11.        substitute $\phi^{GRF}[\text{False}/\text{atomToProposition}(a)]$;
12.   end
13. end
14. return $M^{GRF} = \langle G^{GRF}, \text{ann}^{GRF}, \phi^{GRF} \rangle$;

The algorithm works as follows: We first create a copy of the input model (Lines 1-2) and then compute the backbone of its May formula (Line 3) – the set of propositional variables (negated and not) that follow from it [Marques-Silva et al., 2010]. We treat this computation as a black-box. Next, we look at each Maybe-annotated atom in the partial model: if it is in the backbone, then we annotate it with True and set it to True in the May formula (Lines 6-8). If its negation is in the backbone, then we remove it from the model altogether and set it to False in the May formula (Lines 9-11).

Given the backbone, the algorithm performs linearly in the number of Maybe-annotated atoms. Therefore, the complexity of the algorithm is the complexity of the backbone computation, which is NP-hard [Kilby et al., 2005].

In our example, the partial model in Figure 3.4(c) has the May formula $Dt \land Da \land Kt \land Ka \land Lt \land La \land S1 \land \neg Bt \land \neg Ba \land (Ca \leftrightarrow Ct)$. The backbone computation returns the set $\{Dt, Da, Kt, Ka, Lt, La, S1, \neg Bt, \neg Ba\}$. 
Each of the atoms represented by the variables $D_t, D_a, K_t, K_a, L_t, L_a, S_l$ is annotated with True, and the variables are substituted with True in $\phi$. The atoms represented by $B_t$ and $B_a$ are deleted, and their variables are substituted by False in $\phi$. The resulting partial model is shown in Figure 3.4(a). Its May formula is $\text{True} \land \text{True} \land \text{True} \land \text{True} \land \text{True} \land \text{False} \land \text{False} \land (C_a \leftrightarrow C_t)$ which evaluates to $C_a \leftrightarrow C_t$.

The following theorems state correctness of Algorithm 1.

**Theorem 3.2.** Let $M$ be a partial model and $M^{GRF}$ be a result of applying Algorithm 1. Then, $M \sim M^{GRF}$.

**Theorem 3.3.** If $M^{GRF}$ is a model computed from $M$ using Algorithm 1, then there does not exist a $M' \sim M$ such that $M'$ has fewer Maybe-annotated atoms than $M^{GRF}$.

### 3.3.2 Propositional Reduced Form (PRF)

**Definition 3.10.** A PRF of a partial model $M = \langle G, \text{ann}, \phi \rangle$, is a partial model $M^{PRF} = \langle G^{PRF}, \text{ann}^{PRF}, \phi^{PRF} \rangle$ such that $M \sim M^{PRF}$ and $G^{PRF}$ only has Maybe-annotated atoms.

A partial model can have multiple PRF equivalent models. Algorithm 2 describes how to put a given partial model into its PRF.

**Algorithm 2: PRF Conversion**

**Input:** A partial model $M = \langle G, \text{ann}, \phi \rangle$

**Output:** A PRF equivalent $M^{PRF} = \langle G^{PRF}, \text{ann}^{PRF}, \phi^{PRF} \rangle$ of $M$

1. $G^{PRF} := G$;
2. $\phi^{PRF} := \phi$;
3. **foreach** atom $a$ in $G^{PRF}$ **do**
   4. **if** $\text{ann}(a) = \text{True}$ **then**
      5. $\text{ann}^{PRF}(a) := \text{Maybe}$;
      6. $\phi^{PRF} := \phi^{PRF} \land \text{atomToProposition}(a)$;
   7. **end**
4. **end**
9. return $M^{PRF} = \langle G^{PRF}, \text{ann}^{PRF}, \phi^{PRF} \rangle$;

The algorithm works as follows: We first create a copy of the input model (Lines 1-2). Next, we find all True-annotated atoms and (a) annotate them with Maybe and (b) conjoin their corresponding propositional variable to the May formula (Lines 4-6). The resulting model only contains Maybe-annotated atoms. The algorithm is linear to the number of atoms of the model.

In our example, to compute the PRF of the GRF partial model in Figure 3.4(a), we annotate all True-annotated atoms with Maybe and conjunct their respective propositional variable to the May formula. The resulting PRF partial model is shown in Figure 3.4(b).

The following theorem states correctness of Algorithm 2.

**Theorem 3.4.** Let $M$ be a partial model and $M^{PRF}$ be a result of applying Algorithm 2. Then, $M \sim M^{PRF}$ and $M$ has only Maybe-annotated atoms.
3.4 Constructing Partial Models

In this section, we give an algorithm for constructing a partial model from a set of known alternatives (Algorithm 3).

ALGORITHM 3: Construction of a partial model from a set of concrete models.

Input: Set \( \mathcal{A} \) of concrete models of type \( T \)
Output: A partial model \( M \), such that \( C(M) = \mathcal{A} \).

1. \( G^T_u := \text{GraphUnion}(\mathcal{A}) \);
2. \( \text{MayAtoms}_M := \{ a | a \in G^T_u \land \exists m \in \mathcal{A} \cdot a \notin m \} \);
3. \textbf{foreach} atom \( a \in G^T_u \) \textbf{do}
   4. \quad \textbf{if} \( a \in \text{MayAtoms}_M \) \textbf{then}
      5. \quad \quad \text{ann}(a) := \text{Maybe};
   6. \quad \textbf{else}
      7. \quad \quad \text{ann}(a) := \text{True};
   8. \textbf{end}
9. \textbf{end}
10. \( \phi := \text{False} \);
11. \textbf{foreach} \( m \in \mathcal{A} \) \textbf{do}
   12. \quad \( \phi_m := \bigwedge \{ \text{atomToProposition}(a) | a \in \text{MayAtoms}_M, a \in m \} \land \bigwedge \{ \neg \text{atomToProposition}(a) | a \in \text{MayAtoms}_M, a \notin m \} \);
13. \quad \phi := \phi \lor \phi_m;
14. \textbf{end}
15. \textbf{return} \( M = (G^T_u, \text{ann}, \phi) \)

Construction of partial models from a known set of models is achieved by merging their graphs and annotating the atoms that vary between them by \text{Maybe}. Additionally, the May formula is constructed to capture the allowable configurations of the \text{Maybe} atoms.

We capture this process in Algorithm 3 as follows:

1. To create a partial model \( M \) from a set \( \mathcal{A} \) of classical models of type \( T \), we first create the typed graph \( G^T_u \) of their graph union (cf. Definition 2.4).

2. Next, we find every atom of \( G^T_u \) for which there exists at least one concretization in \( \mathcal{A} \) that does not contain it. We store all these atoms in the set \( \text{MayAtoms}_M \) (Line 2). Every atom in \( G^T_u \) then gets an annotation: if an atom is in \( \text{MayAtoms}_M \), it is annotated with \text{Maybe}; otherwise, it is annotated with \text{True} (Lines 3-8).

3. We construct the May formula \( \phi \) as a disjunction of conjunctions of \text{Maybe}-annotated atoms. We initialize \( \phi \) as \text{False} (Line 10) and iteratively add disjunctive clauses for each model \( m \) in \( \mathcal{A} \). More specifically, given \( m \), we create a conjunction \( \phi_m \) of all \( \text{MayAtoms}_M \). If an atom \( a \) of \( \text{MayAtoms}_M \) does not exist in \( m \), it appears in \( \phi_m \) negated (Line 12). Ultimately, the May formula becomes the disjunction of all the individual \( \phi_m \) clauses (Line 13).

In the PtPP example, the six alternative behavioural designs in Figure 1.4(a-f) can be represented using the partial model shown in Figure 1.3. For example, the state \text{Finishing} exists in only two concretizations, and the transition on \text{restart} in three; therefore, both are represented as \text{Maybe}. Atoms such as the state \text{Idle} are present in all concretizations, and thus are represented as \text{True} (i.e., without the star decoration). The corresponding May formula is shown at the bottom of Figure 1.3.
The algorithm is linear in the total number of atoms and by construction creates a partial model with a set of concretizations equal to the input set of models. We codify this in the following theorem.

**Theorem 3.5.** Applying Algorithm 3 to a set of classical models $\mathcal{A}$ results in a partial model $M$ such that $C(M) = \mathcal{A}$.

The next theorem leads us to observe that Algorithm 3 can be used as a “brute-force” algorithm to compute the GRF of a partial model, by first breaking the partial model down to its constituent concretizations and then merging them again.

**Theorem 3.6.** Models generated in Algorithm 3 are in GRF.

In Section 4.3, we compare the relative efficiency of Algorithms 1 and 3 for putting a partial model into GRF.

### 3.5 Related Work

The notion of design-time uncertainty used in this work is that of “multiple possibilities”. Thus, partial models are a formalism where a set of models is encoded in a single modelling artifact. This is a common abstraction pattern in software engineering, often used in cases where developers are required to work with sets of related and/or similar models. Examples of formalisms that follow the same abstraction pattern are metamodels, partial behavioural formalisms, and software product lines. Since such formalisms implement the same abstraction pattern, and given that some of them have similar expressive powers, it would be tempting to unify them in a single language. However, this would be counter-productive because each of them has been conceived with a different intent in mind. The intent of each formalism defines its assumed context of use and prioritizes what tasks are important for the formalism to efficiently support its context. In the following, we briefly discuss the related formalisms discussing their particular intended context of use, and compare them to partial models.

#### 3.5.1 Metamodels

A *metamodel* is a model that defines the set of models which conform to it, similar to how a grammar defines the set of acceptable phrases of a language. Metamodels are typically expressed using a meta-metamodelling language such as MOF [Object Management Group, 2006a] and KM3 [Jouault and Bézivin, 2006]. Additional constraints are typically expressed in a constraint language such as OCL [Object Management Group, 2006b]. In principle, we could use a metamodel to express the set of possible models from which a team has to choose in order to resolve their uncertainty. This would involve refining the metamodel of the language used for development to one containing very specialized “singleton” types of atoms. Such an approach would be very awkward, as it involves conceptually altering the development language itself, the consequences of which may be hard to manage.

#### 3.5.2 Partial behavioural models

A number of partial *behavioural* modelling formalisms have been studied in the context of abstraction (for verification) or for capturing early design models [Fischbein et al., 2012].
Modelling uncertainty in behavioural models Modal Transition Systems (MTSs) [Larsen and Thomsen, 1988] allow introduction of uncertainty about transitions on a given event, whereas Disjunctive Modal Transition Systems (DMTSs) [Larsen, 1991] add an additional constraint that at least one of the possible transitions must be taken in the refinement. Restricted DMTSs (rDMTSs) [Ben-David et al., 2013] is a variant of DMTSs that is closed under the merge for models of different vocabularies. Incompletely and partially labelled transition systems (IPLTS) allow the definition of states that represent parts of a system that are yet to be fully specified [Ghezzi et al., 2014]. These approaches compactly encode an over-approximation of the set of possible LTSs and thus reasoning over them suffers from information loss. Moreover, they employ refinement mechanisms that allow resulting LTS models to have an arbitrary number of states. This differs from the treatment provided in this paper, where we concentrated only on “structural” partiality and thus state duplication was not applicable. While the intent of behavioural specification methods is the same as ours, partial models are applicable to any kind of modelling language (not just behavioural models) that can be defined using a metamodel. While we have used a state-machine-like language for the motivating example in this presentation, there is nothing inherent in our approach that mandates it. For example, in Section 3.2.2, we applied our approach to a toy custom structural language for describing arrangements of boxes. It has also been applied to languages such as UML Class Diagrams [Famelis et al., 2013] and goal models [Salay et al., 2012a].

Underspecification and Nondeterminism When specifying a system, scope defines what elements are relevant to the system, and span – what level of abstraction is acceptable. Elements that are outside the scope and span of the specification are ones the developer does not care about. Elements that are within the scope and span, but are not present in the specification, are underspecified. In practice, this is often used to indicate that these elements are to be re-visited at a later point when more information becomes available. Thus, an underspecified system represents a set of fully specified possibilities. Nondeterminism is a common way to interpret underspecification in the context of behavioural modelling. Underspecification is used to implicitly (through omission) hint at what developers are uncertain about. On the contrary, partial modelling is used to explicitly define what the developers do not know. In addition, partial modelling allows us to combine underspecification and explicit uncertainty. For example, MTSs allow underspecification using non-determinism, while also having explicit utterances of uncertainty.

3.5.3 Software product lines

Another relevant area is software product line (SPL) engineering [Pöhl et al., 2005] where a product line captures a family of models by identifying their commonalities and variabilities. The notion of a feature is central in variability modelling, albeit informally defined (e.g., in the Feature Oriented Analysis and Design (FODA) approach, a dictionary definition is used: “A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems” [Kang et al., 1990]). The main intent of SPL engineering is therefore the strategic maintenance of a set of modelling artifacts that realize high-level features, and that can be combined to create desirable products. In contrast, the intent of uncertainty modelling is to facilitate decision deferral as discussed in Section 1. In fact, these two intents can co-exist within the same modelling artifact. For example, a practitioner may be uncertain about whether to include a feature in a product line. In the rest of this section we elaborate on the differences.
Modelling variability  Features are defined at an abstract level and are conceptually separate from the software artifacts. Their expression in software models is done using variability points. Dependencies between features are typically maintained in a separate Feature Model [Kang et al., 1990; Schobbens et al., 2006]. Some approaches incorporate expressions of variability directly using notational extensions in the metamodel [Morin et al., 2009]. In practice, however, the annotative approach is typically used, where variability points take the form of presence conditions attached to model elements [Czarnecki and Antkiewicz, 2005; Kästner and Apel, 2008]. These conditions are propositional expressions over the vocabulary of features. They ensure that a model element is only included in a product if the desired combination of features is selected. An example product line, taken from [Classen et al., 2010], is shown in Figure 3.5. This product line represents a family of vending machines and consists of a Feature Model (shown on the left) and a Featured Transition System (shown on the right). A Featured Transition System is a behavioural model, where transitions are annotated with presence conditions. It thus encodes a family of Labelled Transition Systems (LTSs), each one modelling the behaviour of an individual vending machine. For example, in Figure 3.5, the transition from state 1 to state 2 has the presence condition $v \land \neg f$, indicating that it is only present in vending machines that have the feature $v$ (“VendingMachine” in the Feature Model) and not the feature $f$ (“FreeDrinks”).

While partial models have some similarities and are close in expressive power to certain annotative SPL techniques, variability languages are not directly applicable to uncertainty management because of their different intent. Specifically, the intent of a variability model is to express how features, i.e., abstract concepts external to the model, manifest themselves in terms of model elements. On the contrary, the intent of a partial model is to express whether some element itself is present or not. For example, the presence condition of a variability point, such as the transition from state 1 to state 2 in Figure 3.5, is expressed in terms of features. On the other hand, a May element in a partial model is mapped to a propositional variable that encodes whether it is part of the model or not (see the atomToProposition mapping in Section 2.1). This is conceptually closer to the notion of design-time uncertainty, where a developer is uncertain about the inclusion of a specific model element to the model.

Richer variability languages  Some SPL approaches, such as Clafer [Bak et al., 2011] and TVL [Boucher et al., 2010], offer rich languages for expressing variability points, similar in function to the May formula.

Figure 3.5: An example product line, representing a family of vending machines. The product line consists of a Feature Model (left) and a Featured Transition System (right). Elements of the transition system are annotated with presence conditions. Image source: [Classen et al., 2010].
of partial models. Compared to these, we note that the notion of partiality studied in this dissertation, where model atoms can be annotated with *Maybe*, is only one of several kinds of partiality developed in [Salay et al., 2012c] and known as “MAVO”. However, increased expressive power comes at a cost to reasoning (discussed in Chapter 4). We conducted an experimental study [Saadatpanah et al., 2012] of the effectiveness of four automated reasoning engines in the presence of the full spectrum of partiality annotations. We checked properties of randomly generated MAVO models using a Constraint Satisfaction Problem (CSP) solver [Tsang, 1993], a Satisfiability Modulo Theory (SMT) solver [De Moura and Bjørner, 2011], an Answer Set Programming (ASP) solver [Marek and Truszczynski, 1998], and the Alloy Analyzer [Jackson, 2006b], which uses a Boolean Satisfiability Problem (SAT) solver [Eén and Sörensson, 2004]. Compared with earlier scalability results that focused on reasoning with partial models containing only *Maybe* partiality [Famelis et al., 2012], we found that the largest MAVO models that the solvers could reasonably tackle were an order of magnitude smaller and took an order of magnitude longer to solve.

SPL approaches attempt to address different concerns and priorities, such as configuration, asset management, etc. The development of an SPL is an involved, long-term process, since the product line model itself is of great value. When a product line is created, it represents a commitment to a set of products where all members of the set are “desirable”. Instead, a partial model is ultimately a disposable artifact that is systematically refined as more information becomes available. In other words, a partial model represents the situation where at least one member of the set is “desirable”. This difference in intent means that, for partial models, emphasis is put on supporting tasks that address changes in the level of uncertainty. We discuss this in Chapter 6.

The transient nature of partial models requires the methodological and tool support for rapidly expressing uncertainty when it is encountered in the development process. This may involve propagating the appearance and/or resolution of uncertainty across different and potentially heterogeneous models [Salay et al., 2012a]. Since the focus is on decision deferral, partial models need to be first-class development artifacts that can be manipulated throughout the software engineering life cycle wherever a classical model would be expected. We discuss this in Chapter 7.

### 3.6 Summary

In this chapter, we have introduced partial models, a formalism for representing sets of possibilities. Partial models express uncertainty by annotating elements of a base model as *Maybe* while a May formula constrains their possible combinations.

The formal semantics of partial models was given in terms of the set of concretizations, i.e., non-partial models, that they encode. A partial model can be refined to produce a new partial model with a subset of the original concretizations. If refinement results in a singleton set, the resulting partial model can be straightforwardly transformed into a non-partial one.

We also gave an algorithm for constructing a partial model from such a set. Since a given set of concretizations can be expressed in many ways, we have formally introduced two “reduced forms”, namely, Propositional Reduced Form (PRF) and Graphical Reduced Form (GRF), which are used for reasoning and modelling, respectively.

We have also described the notation of partial models, which is used throughout the thesis. We have introduced a partialization process for adapting the metamodel of the base language to support
expressing uncertainty. The main goal that the notation tries to satisfy is to be unambiguously used with any base language, without breaking metamodel conformance rules.

The partial model semantics defined in this chapter is crucial in defining the meaning of the property verification procedure described in Chapter 4, as well as for defining the correctness of transformation lifting, described in Chapter 5. Finally, the reduced forms are used in Chapter 4 to produce useful user feedback, as well as in Chapter 6, where we further elaborate on the pragmatics of refinement in Chapter 6, introducing “property driven refinement”.
Chapter 4

Reasoning in the Presence of Uncertainty

In this chapter, we describe how to perform automated verification in the presence of uncertainty. In particular, we describe the following reasoning operations:

OP1: **Verification**: how to check whether a partial model satisfies a property.

OP2: **Diagnosis**: how to find out which alternatives violate the property.

We discuss a third reasoning operation, Refinement, in Chapter 6.

Faced with a set of alternative solutions to a problem and uncertainty about choosing one of them, developers can create partial models and use them to leverage automated verification techniques. In order to reason effectively about properties of the entire set of alternatives, the developers may want to ask the questions such as:

*Does the property hold for all, some, or none of the alternatives?* This can help determine how critical some property is in selecting alternatives. For example, in the PtPP scenario, the property P3: “Every state has an outgoing transition” holds for all alternatives in Figure 1.4, and therefore is not going to be a main reason in selecting one over the other. Moreover, if some property does not hold for any alternative, it may be an indication that the team needs to revisit the designs, sooner rather than later. For example, knowing early on that P2 (no two transitions have the same source and target) does not hold for any of the alternatives in Figure 1.4(c, d) may be an indication that the team needs to reconsider the design of the “selfish” scenario.

*If the property does not hold for all alternatives, why is it so?* This form of diagnosis can help guide development decisions even before uncertainty is lifted. We further elaborate it into two sub-questions:

*What do alternatives for which the property does not hold look like?* Developers may be interested in finding an alternative that is a counter-example to a property (or if they expected that the property would be violated – an example where it holds) to help them debug the set of alternatives. For example, Figure 1.4(e) is a counter-example to the property P4: “Users can always cancel any operation (i.e., every non-idle state has a transition to Idle on cancel).

*What do alternatives for which the property does not hold have in common?* This form of diagnosis allows developers to explore whether there is a common underlying cause for the violation of the property. To do this we need to calculate a partial model encoding the entire subset of alternatives that violate the
property. For example, out of the six concretizations of $M_{p2p}$, shown in Figure 1.4, the two concretizations implementing the “selfish” scenario, i.e., those in Figures 1.4(c, d), violate the constraint P2. To help developers identify this, we can present to them the partial model shown in Figure 3.4(a), which precisely encodes these two concretizations.

In the following, we build on the formal semantics of partial models, introduced in Chapter 3, to show how to answer these questions. The rest of the chapter is organized as follows: In Section 4.1, we define the semantics of property checking and give a decision procedure for accomplishing it. In Section 4.2, we introduce the different kinds of feedback that can be generated in order to help developers perform diagnosis. In Section 4.3, we describe the experimental evaluation of our approach. In Section 4.4, we discuss related work, and the chapter concludes with Section 4.5.

The contents of this chapter have been published in [Famelis et al., 2012] and expanded in the manuscript [Famelis et al., 2015c], currently under review.

4.1 Checking Properties of Partial Models

4.1.1 Verification semantics

Partial models allow developers to perform reasoning tasks without having to artificially resolve their uncertainty. The idea of using partial modelling in concert with automated reasoning was originally developed for behavioural modelling (finite state machines) [Larsen and Thomsen, 1988]. The semantics of (non-partial) behavioural models is defined in terms of traces, i.e., sequences of transitions or states that are observed during execution. For example, starting from the state $\text{Idle}$, a trace of the state machine in Figure 1.2(b) is the sequence $\{\text{start, completed, share, cancel}\}$. Partial behavioural models, such as Modal Transition Systems (MTSs), were introduced to model the uncertainty of developers about what traces should be included or excluded from a model. Thus uncertainty in partial behavioural models was defined as uncertainty over sets of possible traces. The removal of uncertainty, i.e., refinement, is defined purely in terms of trace preservation, irrespective of the structure of the model. Checking a property of a partial behavioural model is therefore tantamount to checking whether the property holds for all, some or none of the traces which it represents. In other words, in the verification of partial behavioural models, the notion of refinement is central and it relies heavily on language semantics, i.e., traces.

Our approach aims to expand the idea of partial modelling to arbitrary modelling languages. However, while there exists a standard approach for defining modelling language syntax, i.e., metamodeling, there is no such standard approach for specifying semantics. Because of that, we limit our approach to a single, language-independent, syntactic notion of refinement, defined in Section 3.1. This notion of refinement, first introduced in [Salay et al., 2012c] aims at preserving structure as opposed to preserving traces. In other words, refinement, and therefore also verification, in our approach is defined not in terms of traces, but in terms of concretizations, per Definition 3.6. We thus limit partial model reasoning to checking of syntactic properties, such as checking class diagrams for inheritance hierarchy acyclicity, as opposed to e.g., deadlock freedom. In the rest of this chapter by “property” we mean “syntactic property”.

In the following, let $\phi$ be a syntactic property defined in a language such as OCL [Object Management Group, 2006b]. We define the semantics of property checking for partial models as follows:

**Definition 4.1.** The result of checking a property $\varphi$ on a partial model $M$ can be True, False or Maybe.
True means that $\varphi$ holds for all concretizations in $C(M)$, False — that it does not hold for any of them, and Maybe — that it holds for some but not all concretizations in $C(M)$.

In behavioural partial models, where verification is defined in terms of traces, this is known as thorough checking [Bruns and Godefroid, 1999].

As uncertainty gets reduced via refinement, values of properties about which we were certain remain unaffected. This follows from Definition 4.1 and we state it formally:

**Theorem 4.1.** For a property $\varphi$ and two partial models $M_1, M_2$ such that $M_2 \preceq M_1$, if $M_1 \models \varphi$ then $M_2 \models \varphi$, and if $M_1 \not\models \varphi$ then $M_2 \not\models \varphi$.

We note that this theorem also holds for behavioural partial models, albeit with a different definition of refinement.

### 4.1.2 Decision procedure

The purpose of the verification task is to answer the question “Does the property hold for all, some, or none of the alternatives?”.

In order to facilitate reasoning, we put the partial model in Propositional Reduced Form (PRF, see Section 3.3) and appropriately combine its PRF-expressed May formula with the formula representing the property we want to check. A SAT solver is then used to check whether the encoding of the model entails that of the property.

Specifically, the verification engine receives a partial model $M$ that is represented in PRF by the propositional May formula $\phi_M$ and whose base language is subject to well-formedness constraints $\Phi_T$, and a property $p$ expressed as a propositional formula $\phi_p$. From these, we construct the expressions $F^+$ and $F^-$:

$$
F^+(T, M, p) = \Phi_T \land \phi_M \land \phi_p \\
F^-(T, M, p) = \Phi_T \land \phi_M \land \neg \phi_p
$$

(4.1)

We then check satisfiability of the expressions $F^+(T, M, p)$ and $F^-(T, M, p)$, using two queries to a SAT solver, combining the results to determine the outcome of the property on the partial model as described in Table 4.1. For example, if both the property and its negation are satisfiable, then there exists at least one concretization of the partial model where the property holds and another where it does not. Thus, in the partial model the property has value Maybe.

<table>
<thead>
<tr>
<th>$F^+(T, M, p)$</th>
<th>$F^-(T, M, p)$</th>
<th>Property $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>SAT</td>
<td>Maybe</td>
</tr>
<tr>
<td>SAT</td>
<td>UNSAT</td>
<td>True</td>
</tr>
<tr>
<td>UNSAT</td>
<td>SAT</td>
<td>False</td>
</tr>
<tr>
<td>UNSAT</td>
<td>UNSAT</td>
<td>The model is Inconsistent</td>
</tr>
</tbody>
</table>

In the PtPP example, in order to check whether the property P2 ("no two transitions have the same source and target") holds for the partial model $M_{\phi_p}$ in Figure 1.3, we first put $M_{\phi_p}$ in PRF to get the propositional formula $\phi_{M_{\phi_p}}$. Then we express P2 as a propositional formula $\phi_{P2}$, by grounding it over the vocabulary of $M_{\phi_p}$, as described in Section 2.2.1. Checking the property means checking the
Chapter 4. Reasoning in the Presence of Uncertainty

satisfiability of the following formulas:

\[
\begin{align*}
\Phi^+(SM, M_{p2p}, P2) &= \Phi_{SM} \land \Phi_{M_{p2p}} \land \Phi_{P2} \\
\Phi^-(SM, M_{p2p}, P2) &= \Phi_{SM} \land \Phi_{M_{p2p}} \land \neg \Phi_{P2},
\end{align*}
\]

where \(\Phi_{SM}\) are the well-formedness constraints of the (base) State Machine metamodel, shown in Figure 3.3(a). The SAT solver returns one of the two models from Figure 1.4(c, d) as the satisfying assignment for \(\Phi^-(SM, M_{p2p}, P2)\) and one of those in Figure 1.4(a, b, e, f) for \(\Phi^+(SM, M_{p2p}, P2)\). Thus, the value of \(P2\) is uncertain (Maybe) in the model.

4.2 Generating Feedback

If the result of the verification task is False or Maybe, the next step is to do diagnosis, i.e., to answer the question "Why does the property of interest not hold?". Or, conversely, if the outcome is Maybe where it was expected to be False, to answer the question "Why is the property not violated?". Three forms of feedback can be returned:

1. Return one counter-example – a particular concretization for which the property does not hold.
2. Return a concretization where the property does hold.
3. Return a partial model representing the set of all concretizations for which the property does not hold, also called a “diagnostic core”.

The first two answer the question “What do alternatives for which the property does (not) hold look like?”, whereas the third answers the question “What do alternatives for which the property does not hold have in common?”. We describe the three forms of feedback in turn below.

4.2.1 Generating a counter-example

A particular concretization for which the property of interest does not hold is provided “for free” as a by-product of SAT-based verification.

In particular, if the property is False, the SAT solver produces a satisfying assignment for \(\Phi^-(T, M, p)\). This assignment is a valuation for all propositional variables in the formula. We use this valuation to generate the corresponding concretization of \(M\), using Algorithm 4.

For each true propositional variable in the valuation, we construct the corresponding model atom and add it to the counter-example. Because of the naming conventions and the semantics of propositional variables and because \(v\) is also a valuation of the well-formedness constraints \(\Phi_T\), this process always results in a well-formed model. The algorithm is linear to the number of propositional variables.

In our running example, verifying property P4 on the model \(M_{p2p}\) involves checking satisfiability of \(\Phi^-(SM, M_{p2p}, P4)\). This formula is satisfiable, and the SAT solver returns one of the concretizations in Figure 1.4(e, f) as a satisfying assignment, e.g., Idle State=True, ..., Finishing State=True, completed Leeching Idle Transition=False, ..., restart Seeding Leeching=True. We then use Algorithm 4 to reconstruct a model, producing the one in Figure 1.4(f). It has states Idle, Finishing, etc. and transitions from Seeding to Leeching on restart but does not contain the transition from Leeching to Idle on completed.
Algorithm 4: Valuation-to-model conversion.

**Input**: A satisfying assignment $v$ of the formula $F$, where $F = F^- (T, M, p)$ or $F = F^+ (T, M, p)$.

**Output**: A model $m$ of type $T$, such that $m$ is a concretization of the partial model $M = (G_M, ann, \phi_M)$.

1. Initialize $m$ as an empty typed graph $(G, type)$, where $G = (V = \emptyset, E = \emptyset, s, t)$;
2. $trueProps = \{u | u \in graphToPropositionSet(M) \land u = True \text{ in } v\}$;
3. foreach $u \in trueProps$ do
   4. if the name of $u$ matches the pattern $n_{-q}$ then
      5. Add new node $n$ to $V$;
      6. type$(n) := q$;
   7. else if the name of $u$ matches the pattern $e_{-x-y-q}$ then
      8. Add new edge $e$ to $E$;
      9. type$(e) := q$;
     10. $s(e) := x$;
     11. $t(e) := y$;
   12. end
4. end
5. return $m$

4.2.2 Generating an example

A concretization for which the property of interest holds is also a by-product of the verification stage: if the result of checking the property is Maybe, the SAT solver produces a satisfying assignment for the formula $F^+ (T, M, p)$. This valuation is converted to a model following Algorithm 4 and provided to the user.

In the case of verifying P4 for PtPP, the SAT solver returns a valuation that corresponds to one of the concretizations in Figure 1.4(a,b,c,d) as a satisfying assignment to the formula $F^+ (SM, M_{p2p}, P4)$.

4.2.3 Generating a Diagnostic Core

To compute a partial model representing the set of all concretizations for which the property does not hold, we note that they are characterized by the formula $F^- (T, M, p)$. In order to create useful feedback to the user, we use this formula in order to construct a new partial model $M^{\neg p}$ that refines the original model $M$, using property-driven refinement, a process described in Section 6.1.

We briefly illustrate this process (defined formally in Algorithm 5) in PtPP. The concretizations of $M_{p2p}$ that violate P2 are those that satisfy the formula $F^- (SM, M_{p2p}, P2)$, i.e., those in Figure 1.4(c, d). The partial model $M^{\neg p}_{p2p}$ that represents them is shown in Figure 3.4(a). To construct it, we first put the model $M_{p2p}$ in Figure 1.3 in PRF. We then conjunct its May formula with $\neg \phi_{P2}$. After removing all superfluous atoms and simplifying, we end up with the model in Figure 3.4(b). Creating its GRF equivalent results in the model in Figure 3.4(a).

4.3 Experimental Evaluation

In this section, we empirically evaluate the feasibility and the scalability of performing property checking and diagnosis with partial models. In Chapter 6, we describe “property-driven refinement”, i.e., the use of properties to guide the resolution of uncertainty. Since the generation of diagnostic cores is a particular
Table 4.2: Model size categories.

<table>
<thead>
<tr>
<th>Size of Model</th>
<th>S</th>
<th>M</th>
<th>L</th>
<th>XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Atoms</td>
<td>(0,110]</td>
<td>(110, 420]</td>
<td>(420, 1640]</td>
<td>(1640, 6480]</td>
</tr>
<tr>
<td>Exemplar</td>
<td>30</td>
<td>240</td>
<td>930</td>
<td>2550</td>
</tr>
</tbody>
</table>

Table 4.3: Categories of the size of the concretization set.

<table>
<thead>
<tr>
<th>Size of Set</th>
<th>S</th>
<th>M</th>
<th>L</th>
<th>XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Concretizations</td>
<td>(0,10]</td>
<td>(10, 75]</td>
<td>(75, 150]</td>
<td>(150, 300]</td>
</tr>
<tr>
<td>Exemplar</td>
<td>5</td>
<td>50</td>
<td>100</td>
<td>200</td>
</tr>
</tbody>
</table>

use case of property-driven refinement, which is experimentally validated in Section 6.2, here we only show the experimental evaluation of counterexample-based diagnosis.

4.3.1 Experimental setup

We attempted to answer the following research questions:

**RQ1:** How feasible is reasoning with sets of models with the partial model representation in comparison to the classical approach, i.e., with an explicit set of classical models?

**RQ2:** How sensitive are the partial modelling representation and reasoning techniques to the varying degree of uncertainty?

To get answers to RQ1 and RQ2, we set up experiments with parameterized, randomly generated inputs to simulate various settings of realistic reasoning in the presence of uncertainty. More specifically, we ran two experiments, E1 and E2. In experiment E1, we measured the performance of accomplishing task OP1 (verification) with partial models. This involves checking the satisfiability of two formulas $F^+$ and $F^-$, as described in Section 4.1.2. In experiment E2, we measured the performance of accomplishing the same verification task OP1 classically. This requires us to check the property on each individual classical model in the set separately: if the property holds for all (no) models, the result is True (False). If we find one model for which the property does and one for which it does not hold, the result is Maybe.

To answer RQ1, we studied the results of the two experiments and compared the observed runtimes. To answer RQ2, we executed the experiments E1 and E2 with randomly generated experimental inputs that were parameterized to allow for different sizes, both with respect to model size and the size of the set of concretizations.

In addition, we stress-tested our approach to determine the limits of partial model reasoning. In particular, we extrapolated from the size categories in Table 4.2 and Table 4.3 and checked properties (experiment E1) for increasingly larger models.

4.3.2 Experimental inputs

The metamodel of typed models corresponds to additional constraints in their propositional encoding. This makes the problem easier for the SAT solver, as it constrains the search space. We chose to use untyped models for inputs to our experiments, as these are the least constrained and thus the most difficult for the SAT solver.
Table 4.4: Properties used for experimentation.

<table>
<thead>
<tr>
<th>#</th>
<th>Property</th>
<th>“Inspired from”</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A node can have at most one outgoing edge.</td>
<td>“Multiple Inheritance”</td>
<td>[Rad, Y. T. and Jabbari, R., 2012]</td>
</tr>
<tr>
<td>T2</td>
<td>No two edges have the same source and target.</td>
<td>“Parallel Edges”</td>
<td>[Rad, Y. T. and Jabbari, R., 2012]</td>
</tr>
<tr>
<td></td>
<td>(P2 in the PtPP example)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>The graph does not contain self-looping edges.</td>
<td>“Cyclic Composition”</td>
<td>[Van Der Straeten et al., 2003]</td>
</tr>
<tr>
<td>T4</td>
<td>Every node has at least one incoming or outgoing edge.</td>
<td>“Disconnected Element”</td>
<td>[Van Der Straeten et al., 2003]</td>
</tr>
</tbody>
</table>

We considered the following experimental parameters: 1. size of the partial model, 2. size of its set of concretizations, 3. specific property checked, and 4. result of property checking (True, False, Maybe). To manage the multitude of possible combinations of these, we discretized the domain of each parameter into categories.

We defined four size categories, based on the total number of elements (nodes and edges) in the partial model: Small (S), Medium (M), Large (L) and Extra-Large (XL). Based on pilot experiments, we defined ranges of reasonable values for each size category as follows: We defined a “seed” sequence (0,10,20,40,80). The boundaries of each category were calculated from successive numbers of the seed sequence using the formula $n \times (n+1)$. Using the same formula we calculated the representative exemplar of each category by setting $n$ to be the median of two successive numbers in the seed sequence. The ranges of the categories and the selected exemplars for each category are shown in Table 4.2.

In a similar manner, we defined four categories (S, M, L, XL) for the size of the set of concretizations of the generated model. The size of this set reflects the degree of uncertainty encoded in the partial model, so that the category S corresponds to little uncertainty over which alternative to choose, and the category XL corresponds to extreme uncertainty. Based on pilot experiments, we defined reasonable ranges and selected a representative exemplar for each category, as shown in Table 4.3.

To generate realistic models while avoiding a combinatorial explosion of possible setups, we also fixed a few structural parameters of the generated models, using three previously published examples constructed using partial models for guidance: (a) a conceptual model of vehicles, expressed as a class diagram [Salay et al., 2012b], (b) a goal model capturing the early requirements for a meeting scheduler [Salay et al., 2012a], and (c) a sequence diagram modelling the behaviour of electric windows in cars [Salay et al., 2013b]. Based on these, we fixed the edge-to-node ratio to 1.55, and the percentage of Maybe-annotated atoms to 8%.

We did experiments with four properties, inspired from well-formedness properties found in the literature and adapted for untyped graphs. The four properties are described in Table 4.4. We illustrate how they were used together with the randomly generated models in Appendix B.

### 4.3.3 Implementation

We implemented tooling support to randomly generate inputs based on the experimental properties outlined in Section 4.3.2. Specifically, we generate EMF [Steinberg et al., 2009] models which are then exported to SMT-LIB [Barrett et al., 2010], the input format of the Z3 solver [De Moura and Bjørner, 2011], using a Model-to-Text transformation. We also use the EMF representation to generate a random set of concretizations for the model, as well as to ground the input property. The corresponding formulas
are then exported to SMT-LIB. We chose the Z3 solver for reasoning because it was the winner of the 2011
SMT Competition [SMT-COMP’11, accessed 12-01-2012]. Additional details regarding the generated
inputs can be found in Appendix B.

4.3.4 Methodology

We conducted a series of experimental runs generating inputs along the dimensions specified by three
parameters: model size, size of set of concretizations, and type of property. For each combination of the
parameters, we produced inputs using the selected exemplar values shown in Tables 4.2 and 4.3.

In order to run experiment E2 (“classical reasoning”), we converted each generated partial model as
follows: (1) we created a set containing all its concretizations; (2) for each one of the concretizations,
we created the SMT-LIB encoding as described in Appendix B and checked to see whether it satisfies
the property.

For E2, we reported the following times: if the property is True or False for the entire set, the time
required to check the property for all concretizations; if the property is Maybe, the time required to find
at least one concretization for which it holds and one for which it does not.

We integrated all the above in an experimental driver built on top of MMINT, a multi-modelling
environment created at the University of Toronto [Di Sandro et al., 2015]. Our experimental driver
created all combinations of input parameters, generated models, translated models and formulas to the
SMT-LIB encoding described in Appendix B and made the appropriate method calls to execute each
experiment. For each combination of input parameters, we repeated the experiment 20 times to generate
a confidence interval smaller or equal to 20% of the mean of the observations. The confidence intervals
were computed at the 95% confidence level, using Student’s t-distribution function. We then categorized
the runs according to the produced results with respect to the possible return values (True, Maybe, and
False).

For each run, we recorded the observed run-times and calculated the speedup \( S_p = \frac{T_c}{T_{pm}} \), where \( T_c \)
and \( T_{pm} \) were the times to do a task with sets of classical models and with partial models, respectively.

For stress testing, we extrapolated from the size categories in Tables 4.2 and 4.3, checking the property
P2 from Table 4.4 for increasingly larger models. We set a maximum timeout of 10 minutes for the two
checks \( (F^+(T, M, p)) \) and \( F^-(T, M, p) \) required for experiment E1 and increased the number of elements
and the number of concretizations until we reached sizes for which the checks timed out. We picked the
timeout duration based on the maximum observed times of E1 runs (see next section).

The setup described above resulted in 320 samples for each of the 4 properties, for a total of 1280
samples, on which we ran E1 and E2. We ran all experiments on a computer with Intel Core i7-2600
3.40GHz×4 cores (8 logical) and 8GB RAM, running Ubuntu-64 12.10.

4.3.5 Results

The experiments\(^1\) did not show dramatic differences in speedup between the different properties, whereas
we observed important differences between the different verification results. The observed verification
results for each property are shown in Table 4.5.

Averaged over all properties, the biggest difference in speedup was recorded between S models with
M sets of concretizations evaluating to Maybe (0.46) and S models with XL sets of concretizations

\(^1\)All results are available at http://www.cs.toronto.edu/~famelis/datasets.html
Table 4.5: Verification results per property.

<table>
<thead>
<tr>
<th>Property</th>
<th>% Maybe</th>
<th>% False</th>
<th>% True</th>
<th>% False or True</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1.9</td>
<td>98.1</td>
<td>0</td>
<td>98.1</td>
</tr>
<tr>
<td>P2</td>
<td>39.7</td>
<td>32.2</td>
<td>28.1</td>
<td>60.3</td>
</tr>
<tr>
<td>P3</td>
<td>22.5</td>
<td>22.8</td>
<td>54.7</td>
<td>77.5</td>
</tr>
<tr>
<td>P4</td>
<td>12.8</td>
<td>87.2</td>
<td>0</td>
<td>87.2</td>
</tr>
</tbody>
</table>

evaluating to True or False (42.90). Averaged over all properties, the largest speedup recorded for checks evaluating to Maybe was for S models with L sets of concretizations (1.48), whereas the smallest speedup recorded for checks evaluating to True or False was for S models with XL sets of concretizations (0.94).

The ranges of speedups for checks evaluating to True/False and those evaluating to Maybe are shown in Figure 4.1(a) and Figure 4.1(b), respectively. The plotted values are averages over all properties for each combination of model size and the size of the set of concretizations. This indicates that these parameters are the most important factors for studying the effectiveness of partial model reasoning.

For checks evaluating to True or False, Figure 4.1(a) shows that there is a significant speedup from using partial models compared to the classical approach. The smallest speedups were observed in the inputs with S sets of concretizations (between 1.68 for S-sized models and 0.94 for XL-sized models). The increase from these values was dramatic for M, L and XL sets of concretizations. For these categories, the smallest speedup was 6.87 for L-sized models with M sets of concretizations and the biggest speedup was 42.90 for S-sized models with XL sets of concretizations.

For checks evaluating to Maybe, Figure 4.1(b) shows a more erratic pattern, with no clear trends for the categories of model size and size of set of concretizations. Most observed speedups are below 1 (except S models with L sets of concretizations (1.48) and models of size M with sets of concretizations of size M (1.29)), and the average speedup for all checks evaluating to Maybe was 0.74, with a low of 0.46 for S models with M sets of concretizations.

Overall, we observe a relative slowdown for checks evaluating to Maybe but a dramatic speedup for checks evaluating to True and False. In addition, we note that in the worst case, the “classical” approach behaves as poorly as for True and False checks. We summarize the observations for all return types in Figure 4.1(c).

Based on the summarized observations and taking into account the trade-off between evaluation return types, we conclude that, regarding RQ1 (feasibility), there is a significant overall net gain from using the partial modelling approach.

Regarding RQ2 (sensitivity to degree of uncertainty), the observations in Figure 4.1(c) point to the conclusion that the speedup offered by our approach is positively correlated to the degree of uncertainty. In fact, the greatest speedups were observed for inputs with the largest sets of concretizations. For smaller levels of uncertainty and for properties that evaluate to Maybe, explicitly handling the set provides a relative speedup.

Our stress tests showed that the solver successfully checked partial models with up to 10000 atoms and 800 concretizations within the 10 minute timeout. It timed out at the next increment, namely, a partial model with 11125 atoms and 900 concretizations.
(a) Checks evaluated to True and False.

(b) Checks evaluated to Maybe.

(c) Average of all checks.

Figure 4.1: Checking properties with partial models (E1) compared to classically (E2). Speedup ($\frac{E_2}{E_1}$) versus model size for different degrees of uncertainty.
4.3.6 Threats to validity

The most important threat to validity of our results stems from the use of experimental inputs that were randomly generated. We attempted to mitigate this threat by using realistic parameters from previously published example partial models, as described in Section 4.3.2.

Additionally, we conducted experiments with only a few input properties. We attempted to mitigate this threat by selecting properties that were inspired from real well-formedness properties, published in the literature (cf. Table 4.4).

Another threat to validity is induced by our choice to use a few exemplar values of the experimental parameters in order to manage the combinatorial explosion of options. Generalizing our experimental observations would require further, more fine-grained experimentation.

To compensate for these threats to validity, we additionally applied our technique to a large non-trivial example based on fixing a bug of the UMLet editor [Auer et al., 2003]. The example, described in Section 8.1, helped us triangulate our experimental results with experience from applying our technique to a real world application. The size of the partial model that we extracted from UMLet, as well as its set of concretizations, fell in the M category. The properties checked where “forall-exists”, similar to Property 4 in Table 4.4 and returned True and Maybe. The observed speedups were even higher than the experimental results presented in Section 4.3.5. This suggests that the experimental results obtained from reasoning with randomly generated models may in fact constitute a lower bound, and that reasoning with real models is even more efficient.

4.4 Related Work

In Section 3.5, we explained that partial models are an example of a common abstraction pattern where multiple software artifacts are encoded in a single representation. We described three other formalisms that follow the same pattern: metamodels, partial behavioural models, and software product lines. In this section, we discuss work related to performing automated reasoning with such formalisms. Additionally, we discuss related work from the field of database management.

**Reasoning with metamodels.** Metamodels are used to define the abstract syntax of modelling languages, and thus a metamodel encodes the set of possible instance models that conform to it. In the context of a multi-tier instantiation hierarchy, such as MOF [Object Management Group, 2006a], this applies by extension to any model that can be instantiated. Reasoning about instances of metamodels is typically done using constraint languages such as OCL [Object Management Group, 2006b]. Well known systems that support verification using OCL include the Dresden OCL Toolkit [Demuth et al., 2009] and USE [Gogolla et al., 2007]. Reasoning about models and metamodels is also done by leveraging reasoning engines such as Alloy [Anastasakis et al., 2007] and Constraint Satisfaction Problem (CSP) solvers [Cabot et al., 2007]. Partial model reasoning subsumes techniques for reasoning about models and metamodels, since they can be used as part of its decision procedure instead of a SAT solver. For example, in [Saadatpanah et al., 2012] we studied partial model reasoning using a variety of reasoning engines, including Alloy and CSP.

**Reasoning with partial behavioural models.** We discussed the fundamental difference in semantics between partial behavioural model verification and our approach in Section 4.1. In short, our approach
is a language independent generalization of partial behavioural modelling. A corollary to language independence is that the approach is only directly applicable for those analysis tasks that do not require knowledge of the underlying semantics of the language. In order to apply to partial models, an analysis technique that needs language semantics must first be “lifted” to accommodate partiality.

Lifting consists of adapting an existing decision procedure to make it uncertainty-aware, i.e., to give it the capability to handle compact representations of sets of possibilities. A number of analyses have been lifted to partial behavioural models: model-checking [Fischbein et al., 2012], pairwise compatibility checking [Krka et al., 2014], and incremental refinement [Uchitel et al., 2013]. The concept of lifting is not limited to reasoning. For example, in Chapter 5, we lift graph rewriting-based model transformations. We are interested in developing a generic technique which can lift an arbitrary procedure, e.g., to make it uncertainty-aware. This has been done, e.g., for reasoning with models of early requirements [Horkoff et al., 2014].

In addition to property checking, the generation of artifacts similar to diagnostic cores, through a process of property-driven refinement (see Chapter 6) has also been studied for partial behavioural modelling [Uchitel et al., 2009], although with a method which is somewhat different from ours: properties are synthesized as MTSs and then merged with the MTS of the system.

**Reasoning with variability.** There has been extensive research in adapting software engineering techniques to make them *variability-aware*. Variability-aware techniques allow working at the level of product lines rather than with explicit sets of individual products, and include techniques for model checking [Classen et al., 2010], type checking [Kästner et al., 2012], testing [Kästner et al., 2012], graph transformation [Salay et al., 2014], and others [Midtgaaard et al., 2014; Thüm et al., 2012]. Such techniques are similar to our approach to reasoning with a set of possibilities at a high level of abstraction. As discussed in Section 3.5, the main differences between partial model reasoning and these techniques are (a) the different level of granularity in which sets are encoded, and (b) the different pragmatics and context of usage between the two approaches.

**Reasoning with databases.** While models are used to create abstractions of software systems, databases are used to abstract, store and manage data. The notion of uncertainty discussed in this thesis is conceptually related to the management of incomplete databases, using possible worlds theory [Abiteboul et al., 1987]. A well-known method for representing uncertainty in databases is the tableau representation [Aho et al., 1979]. In fact, work on using tableaux to reason about multiple repairs to inconsistency [Wijsen, 2005] was one of the initial inspirations for our work on partial models. The diagnostic techniques presented in this chapter are also closely related to work in the database theory, specifically in the context of data provenance [Buneman and Tan, 2007]. A typical data provenance problem is query inversion, i.e., determining which tuples contributes to the output of a query [Buneman et al., 2000].

### 4.5 Summary

In this chapter we discussed how partial models can be used to accomplish verification tasks in the presence of uncertainty. We defined the meaning of partial model property checking and compared it to partial behavioural model verification, which was the original inspiration for our approach. We ex-
plained that because the partial models presented in this thesis have concretization-based semantics (i.e., are defined through structure-preserving refinement), we scoped our verification technique to checking syntactic properties.

We gave a decision procedure for checking such properties and described the kinds of feedback that can be generated in order to help developers perform diagnosis of their partial models. Specifically, we described three kinds of diagnostic feedback: generating a counterexample concretization, generating an example concretizations where the property holds, and generating a diagnostic core, i.e., a partial model encoding the subset of concretizations that violate the property.

Finally, we performed an experimental evaluation of our verification approach. The goal of the experiments was to assess the feasibility and scalability of property checking and counterexample generation. We used randomly generated model inputs, on which we verified four properties from the literature using a SAT solver. We found that partial model verification results in significant speedup compared to classical verification (i.e., checking the property on each concretization separately). We also observed that the speedup increases with greater degrees of uncertainty (i.e., larger sets of concretizations).

We describe how the results of property checking and diagnosis are used to guide the resolution of uncertainty through property-driven refinement in Chapter 6. We also discuss how partial model verification fits within the overall context of Model-Driven Engineering methodology in Chapter 7.
Chapter 5

Transforming Models Containing Uncertainty

Model Driven Engineering (MDE) promises to accelerate and improve the quality of software development: software is described using high-level models which are easy to reason with; these models are then transformed into lower-level designs through a series of model transformations. Finally, low-level designs are used for effective code generation. However, existing model transformation solutions do not handle models with uncertainty. So when uncertainty is unresolved, the modeller should either delay the application of transformations until more information becomes available, or make premature resolutions in order to apply transformations, thus creating a risk that these resolutions are incorrect. In either case, uncertainty diminishes the benefits of MDE.

In this chapter, we propose an approach that allows applying existing transformations to models containing uncertainty. The essence of the approach involves automatically modifying – “lifting” – transformations so they operate on partial models and correctly transform both the content of the model and the uncertainty about it. As a result of our approach, transformations can be applied early in the model development lifecycle, tolerating the uncertainty and allowing modellers to defer its resolution until extra information is available. This eliminates the need to delay transformation application and removes the pressure to potentially compromise model quality by resolving the uncertainty prematurely.

The rest of the chapter is organized as follows: In Section 5.1, we introduce the intuition of lifting using a new running example, called SolvEx. SolvEx is smaller than the PtPP example introduced in Chapter 1, thus allowing us to better illustrate lifted transformation application; it is also an example of a structural, rather than a behavioural, modelling scenario, further illustrating the language independence of partial modelling. In Section 5.2, we give the formal definition of lifting and describe lifted transformation application. In Section 5.3, we describe the kinds of transformation properties preserved by lifting. We discuss related work in Section 5.5 and conclude in Section 5.6.

The contents of this chapter have been published in [Famelis et al., 2013].

5.1 Intuition of Lifting

To motivate and illustrate the key points of our approach, we introduce a new example called SolvEx, where a modeller is creating a UML class diagram for an automated reasoning engine. In SolvEx,
the modeller has decided that there should exist a class `Solver` which throws exceptions, objects of type `SolverException`, whenever it reaches an error state. However, the modeller has yet to make the following design decisions: (a) whether `SolverException` should be an inner class of `Solver`, and (b) whether `SolverException` should have a `String` attribute called `effect` that would record an estimation of the effect of the exception on the reasoning process. In addition, the modeller expects that at least one of these features will be present in her model.

The resulting UML class diagram with uncertainty is encoded as a partial model $M_1$ in Figure 5.1(a). In this model, “$\dagger$” is the UML symbol for a “nested class” and used here to indicate that `SolverException` is an inner class of `Solver`. This model has two points of uncertainty — the relationship between the classes `Solver` and `SolverException` (denoted by $x$) and the presence of the `effect` attribute in class `SolverException` (denoted by $y$).

The modeller interacts with her tooling using the concrete UML syntax, i.e., the diagram in Figure 2.5(a). However, for the purpose of any automated task the model $M_1$ is used in its abstract syntax. In the rest of this chapter, we only consider models expressed in a simplified version of the UML abstract syntax, defined by the metamodel in Figure 2.5(d). To better illustrate the details of each point of uncertainty in $M_1$, we show the model as a typed graph in Figure 5.1(b). In it, the node $e$ and the edges $sn$, $eo$, $et$, $en$, $ept$ are annotated with `Maybe` and thus indicated using the star icon. We have also included the type of each edge alongside the `Maybe` annotation. The May formula $\Phi_1$, also shown in Figure 5.1(b), constrains the possible combinations of `Maybe` elements, defining the three concretizations $m_{11}, m_{12}, m_{13}$ shown in Figures 5.1(c-e), respectively.

Assume that the modeller notices that her model has an anti-pattern, namely, that the attribute `effect` is public. She decides that, unless `SolverException` is an inner class, `effect` should be made private for security reasons, and be accessed through a getter method. This can be accomplished by performing the `EncapsulateVariable` refactoring [Casais, 1994]. A generic method for implementing this refactoring using graph transformations was described by Mens et al. [Mens et al., 2005]. A simplified version of this rule, called $R_{EV}$, is shown in Figure 5.2. The left-hand side (LHS) of the rule matches a node $a$ (and its associated edges such as $aoowner$) that represents a public attribute. The right-hand side (RHS) makes it private (by deleting the `isPublic` edge $apt$ from $a$ to `True` and adding a new `isPublic` edge $apf$ from $a$ to `False`). It also creates a public getter operation $ge$ and its associated

---

Figure 5.1: (a) partial model $M_1$, showing points of uncertainty. (b) $M_1$ as a typed graph (UML abstract syntax). (c-e) Concretizations $m_{11}, m_{12}, m_{13}$ of $M_1$. 
Figure 5.2: Transformation rule $R_{EV}$ for doing the Encapsulate Variable refactoring.

Figure 5.3: (a-c) Classical models resulting from applying $R_{EV}$ to the classical models in Figure 5.1(c-e). (d) The result of applying the lifted version $R_{EV}$ to $M_1$ directly: the partial model $M_2$.

Rule $R_{EV}$ cannot be directly applied to the partial model $M_1$ using the classical technique, described in Section 2.3, because of uncertainty. Our goal is thus to create its “lifted” version, $R_{EV}$ that can be applied directly to $M_1$. The intuition behind such a lifting is as follows: take the three concretizations, $m_{11}$, $m_{12}$, $m_{13}$, of $M_1$; classically apply $R_{EV}$ to each of them, resulting in models $m_{21}$, $22$, $m_{23}$ in Figure 5.3(a-c); represent the resulting models as a partial model $M_2$ in Figure 5.3(d). That is, applying the lifted rule to a partial model should be equivalent to a representation of the result of applying the original rule to each of the concretizations of the partial model. So, applying the lifted version $R_{EV}$ of $R_{EV}$ to $M_1$ should produce the partial model $M_2$ directly, without having to produce and transform individual concretizations.

5.2 Lifting Algorithm

In this section, we describe the process of lifting a transformation rule to apply to partial models. We focus on transformations based on graph rewriting [Ehrig et al., 2006], as described in Section 2.3.
5.2.1 Specification of lifting

A classical rule $R$ adapted to apply to partial models is called lifted and is denoted by $\mathcal{R}$.

Partial models are intended to be exact representations of sets of models and lifted transformations should preserve this. Therefore, applying a lifted transformation rule $\mathcal{R}$ to a partial model $M_{in}$ should be equivalent to applying its classical version $R$ to each of the concretizations of $M_{in}$ and building a partial model from the result. We refer to this principle as the Correctness Criterion for lifting transformations and define it formally below.

**Definition 5.1.** Let a rule $R$, a partial model $M_{in}$ with a set of concretizations $\left[M_{in}\right] = \{m_{in}^1, \ldots, m_{in}^n\}$, and the set $U = \{m^i_{out} \mid \forall m^i_{in} \in [M_{in}] : m^i_{in} \xrightarrow{R} m^i_{out}\}$ be given. $\mathcal{R}$ is a correct lifting of $R$ iff for any production $M_{in} \xrightarrow{\mathcal{R}} M_{out}$, the set of concretizations of the resulting partial model $M_{out}$ satisfies the condition $\left[M_{out}\right] = U$.

In the SolvEx example, we aim to compute a lifted version $\mathcal{R}_{EV}$ of $R_{EV}$ such that for the partial models $M_1$ and $M_2$ in Figures 5.1(b) and 5.3(d), $M_1 \xrightarrow{\mathcal{R}_{EV}} M_2$.

In classical rule application during graph transformation (described in Section 2.3) it is sufficient to find a graph match of the LHS of the rule and then check whether the NACs are applicable. However, a partial model also has a propositional component, the May formula which constrains the possible combinations of Maybe elements. Thus, doing the graphical match for the May graph is not sufficient to guarantee correctness and needs to be augmented with manipulation of the May, to ensure that the appropriate concretizations get transformed. We illustrate both parts of the transformation on the SolvEx example in Section 5.2.2 and then generalize in Section 5.2.3. In Section 5.2.4 we prove correctness of this approach.

5.2.2 Lifting example

We illustrate the transformation of the graph and the formula using the SolvEx example where the rule $R_{EV}$ in Figure 5.2 is applied to the partial model $M_1$, shown in Figure 5.1(b), to produce $M_2$ in Figure 5.3(d). Figure 5.4 summarizes the application of this rule for the single existing matching site, showing the diagrams and the truth tables of the May formulas $\Phi_1$ and $\Phi_2$ of $M_1$ and $M_2$, respectively. Each column of the truth tables is a Maybe element. Each row corresponds to an allowable configuration of Maybe elements, denoted by 1, and thus defines a concretization. The truth tables also show which Maybe elements are matched by each of the rule’s parts (the parts of $R_{EV}$ are shown in Figure 2.6). For example, the edges $eo$ and $ept$ are both matched by the rule’s LHS, where $eo$ is found in the match $C'$ of the $C'$ part of the rule, and $ept$ in $D'$ match of $D'$. Our objective is to construct the lifted transformation $\mathcal{R}_{EV}$ that produces $M_2$ when applied to $M_1$. We begin by constructing the graphical part of $\mathcal{R}_{EV}$ first, followed by the propositional part.

**Graphical part** Consider applying $R_{EV}$ to $M_1$ by directly applying it to $M_b$, $M_1$’s base graph. Clearly, this approach does not produce the correct outcome. First, NAC$1$ matches in $M_b$ and thus the rule does not apply at all! Yet, there exists a concretization of $M_1$, $m_{13}$, for which neither of $R_{EV}$’s NACs match and thus $R_{EV}$ should be applicable. We therefore expect it to be applicable to $M_1$ as well. Second, the RHS of the rule does not specify which elements in the output model should become Maybe, whereas $M_2$ clearly has them.
Thus, the classical strategy for rule application is not sufficient and needs to be augmented by the uncertainty in the model, i.e., the `Maybe` annotations of its elements. The presence of `Maybe` elements in the match of NAC\textsubscript{1} and in the match of the LHS of R\textsubscript{EV} are both indications that R\textsubscript{EV} applies to some concretizations but not others. We thus need to change the partial model M\textsubscript{1} so that it represents both those concretizations that are unchanged by R\textsubscript{EV} and those where the rule has been applied. Applying R\textsubscript{EV} to a concretization of M\textsubscript{1} entails (1) deleting the edge ept, because it is included in the match D of D\textsubscript{r}, and (2) adding the elements of A: ge, gn, go, gt, gp, and epf. However, we cannot altogether delete ept from M\textsubscript{1} because it should still remain in the unchanged concretization m\textsubscript{11}. Instead, we must keep it annotated with `Maybe` to indicate that it is part of some concretizations but not others. Similarly, the newly added elements should be annotated with `Maybe` to indicate that they are added in m\textsubscript{13} but not in M\textsubscript{11} or m\textsubscript{12}.

We summarize the application of the graphical part of R\textsubscript{EV} to M\textsubscript{1} as follows: (a) Apply R\textsubscript{EV} to the base graph M\textsubscript{b} of M\textsubscript{1} even though NAC\textsubscript{1} matches because the match contains a `Maybe` element. (b) Include both D and A in the base graph of M\textsubscript{2} and annotate all of their elements with `Maybe` because the match of the LHS in M\textsubscript{1} contains a `Maybe` element.

**Propositional part** We now define the propositional part of R\textsubscript{EV} that transforms the May formula Φ\textsubscript{1} of M\textsubscript{1} into Φ\textsubscript{2} of M\textsubscript{2}. We achieve this by defining an operation on Φ\textsubscript{1} that has the effect of transforming the truth table of Φ\textsubscript{1} into the truth table of Φ\textsubscript{2}. First note that the truth tables can be split into two parts:
(a) The concretizations where $R_{EV}$ does not apply (i.e., $m_{11}$ and $m_{12}$). The corresponding rows $m_{21}$ and $m_{22}$ in $\Phi_2$ remain unchanged, and the variables in $A$ are set to False (denoted by 0) to indicate that $A'$ is not added. We denote the formula representing the unchanged part by $\Phi_{nchg}$.

(b) The concretizations where $R_{EV}$ does apply (i.e., $m_{13}$). The corresponding row $m_{23}$ has the variables of $D$ set to 0 to indicate that $D'$ is deleted and the variables of $A$ set to 1 to indicate that $A'$ is added. We denote the formula representing the changed part by $\Phi_{chg}$.

Thus, $\Phi_2 = \Phi_{nchg} \lor \Phi_{chg}$.

To obtain the unchanged part, we begin by specifying a condition, over elements of $M_1$, under which the rule $R_{EV}$ applies, i.e., when its NAC $N$ does not match in $M_b$ and both $C$ and $D$ do match: $\Phi_{apply} = \neg \phi^{and} \land \phi^{and} \land \phi^{and}$. Let $\phi^{and}_X$ where $X \in \{N,C,D\}$ denote the conjunction of all variables in $X$ that represent elements that are Maybe. Restricting $\Phi_1$ to those concretizations of $M_1$ where $R_{EV}$ does not apply ($\neg \Phi_{apply}$) and forcing the variables of $A$ to become False produces the unchanged part: $\Phi_{nchg} = (\Phi_1 \land \neg \Phi_{apply}) \land \neg \phi^{or}_A$, where $\phi^{or}_A$ indicates the disjunction of all variables in $A$ that represent elements that are Maybe.

For the changed part $\Phi_{chg}$, we restrict $\Phi_1$ to those concretizations of $M_1$ where $R_{EV}$ does apply and force the variables of $D$ to become False and those of $A$ to become True: $\Phi_{chg} = (\Phi_1 \land \Phi_{apply}) \land \neg \phi^{or}_D \land \phi^{and}_A$. Here, $(\Phi_1 \land \Phi_{apply}) \land \neg \phi^{or}_D$ indicates existential quantification of all variables in $D$ that occur in formula $\Phi_1 \land \Phi_{apply}$. In our example, $D = \{\text{ept}\}$, so it becomes $(\Phi_1 \land \Phi_{apply})|_{\text{ept}=T} \lor (\Phi_1 \land \Phi_{apply})|_{\text{ept}=F}$.

That is, we eliminate each variable in $D$ from $\Phi_1 \land \Phi_{apply}$ by taking the disjunction of the cases where it is set to False and to True. Quantifying out variables in $D$ is done before forcing them to become False (using $\neg \phi^{or}_D$) because we are changing the values of existing variables (the variables of $D$ already occur in $\Phi_{apply}$) and not just setting the value for new variables as we are for $A$. Otherwise, we get an inconsistency because $\Phi_{apply} \Rightarrow \phi^{and}_D$ by definition.

Substituting the variables from the example and simplifying gives:

\[
\phi_{nchg} = (\text{sn} \land \text{e} \land \text{eo} \land \text{et} \land \text{en} \land \text{ept} \land \neg \text{epf} \land \neg \text{ge} \land \neg \text{go} \land \neg \text{gt} \land \neg \text{gp} \land \neg \text{gn}) \lor \\
(\text{sn} \land \neg \text{e} \land \neg \text{eo} \land \neg \text{et} \land \neg \text{en} \land \neg \text{ept} \land \neg \text{epf} \land \neg \text{ge} \land \neg \text{go} \land \neg \text{gt} \land \neg \text{gp} \land \neg \text{gn})
\]

\[
\phi_{chg} = (\neg \text{sn} \land \text{e} \land \text{eo} \land \text{et} \land \text{en} \land \neg \text{ept} \land \text{epf} \land \text{ge} \land \text{go} \land \text{gt} \land \text{gp} \land \text{gn})
\]

The resulting formula $\Phi_2 = \Phi_{nchg} \lor \Phi_{chg}$ is the same as the May formula in Figure 5.3(d) and has the same truth table as the one shown in Figure 5.4.

### 5.2.3 General case

We can generalize the above process to an arbitrary rule $R$ and define how the graphical part of its lifted version $R$ is applied to a partial model $M$ to produce a partial model $M'$. As with the SolvEx example, we define this in terms of applying $R$ to the base graph of $M$ and then making modifications. Following Definition 2.7, the matching site for $R$ is a matching site for $R$ in the base graph of $M$.

**Definition 5.2** (Lifted rule applicability conditions). Given a partial model $M$ with a May formula $\Phi_M$, a transformation rule $R = \langle \{NAC\}, \text{LHS}, \text{RHS}\rangle$, and a matching site $K = (N,C,D)$, the lifted rule $R$ is applicable at $K$ iff the following conditions hold:

1. For all $N \in N$, $N$ contains a Maybe element
2. $\Phi_M \land \Phi_{apply}$ is satisfiable, where $\Phi_{apply} = \neg \bigvee\{\phi_N|N \in N\} \land \phi^{and}_C \land \phi^{and}_D$. 


In this definition, Condition 1 ensures that there is no NAC match without Maybe elements; otherwise the NAC match would necessarily occur in every concretization and so \( R \) would not apply to any concretization of \( M \). Condition 2 uses the constraints in the May formula to ensure that at this matching site, the rule \( R \) matches in at least one concretization of \( M \). Specifically, this checks that there exists a concretization in which all of the Maybe elements of \( C \) and \( D \) are True and not all of the Maybe elements in any NAC match are set to True.

We now give the general definition of a rule application for a lifted rule.

**Definition 5.3** (Lifted rule application). Assume a partial model \( M \) with a May formula \( \Phi_M \), a transformation rule \( R = \langle \{NAC\},LHS,RHS \rangle \) and matching site \( K = \langle \mathcal{N},C,D \rangle \) in \( M \) for which the rule applicability conditions are satisfied. We define the lifted application of \( R \) as the following process to apply the lifted rule \( R \) to \( M \) to produce a partial model \( M' \):

1. if \( K \) contains no Maybe elements, apply \( R \) in the classical way to produce the base graph of \( M' \) and set \( \phi_{M'} = \phi_M \).
2. otherwise,
   (a) set \( M' = M \);
   (b) add the elements \( A \) of the \( A^r \) part of the RHS to \( M' \);
   (c) annotate all elements of \( A \) and \( D \) with Maybe;
   (d) set \( \Phi_{M'} = [ (\Phi_M \land \neg \Phi_{apply}) \land \neg \phi^A_X ] \lor [ (\Phi_M \land \Phi_{apply}) ] \land I^D_D \land \neg \phi^A_X \land \phi^{and}D ] \).

In the following we refer to lifted application simply as application.

In this definition, Case 1 captures the situation when there are no Maybe elements at the matching site and so the rule can be applied in the classical way and the partial model is unaffected. Case 2 captures the situation when there are Maybe elements in parts of the matching site so that \( R \) may apply in some concretizations but not in others. This case mirrors the discussion of \( R_{EV} \) in Section 5.2.2. In particular, in the graphical part, \( D^r \) is not deleted (step a) but \( A^r \) is still added (step b) and all of the elements in \( A \) and \( D \) are set to Maybe (step c). The propositional part (step d) is the same as for the \( R_{EV} \) example except that \( \Phi_{nchg} \) and \( \Phi_{chg} \) are inlined and the more general case of \( \mathcal{N} \) is used in \( \Phi_{apply} \) (from Definition 5.2) to account for multiple NAC matches that could exist in the base graph of \( M \).

As with a classical rule system, lifted rules continue to be applied until no rule is applicable. Note that the resulting model \( M' \) may not necessarily be in GRF after every rule application. That is, \( M' \) can contain redundant Maybe elements. If \( M' \) is intended for human consumption (as opposed to automated reasoning) then the additional step of putting it into GRF is advisable. However, this step is optional since it does not affect the set of concretizations that the partial model represents.

**5.2.4 Correctness**

We now show that lifting described by Definitions 5.2 and 5.3 satisfies the correctness condition in Definition 5.1. Specifically, we prove the following theorem:

**Theorem 5.1.** Lifted application of a rule \( R \) produces a correct lifting of \( R \).

In other words, if the lifted version \( \mathcal{R} \) of a rule \( R \) is applied to a partial model \( M \) to produce a partial model \( M' \), then the concretizations of \( M' \) must be exactly the set obtained by applying the classical rule \( R \) to each concretization of \( M \). We focus our argument on a specific matching site since by transitivity, if the rule is correct when applied to each site, then the application to any sequence of sites is also correct.
We begin by checking correctness of the applicability condition (see Definition 5.2) of the lifted rule \( \mathcal{R} \): whenever \( R \) is applicable for some concretization of \( M \) at \( K \), then \( \mathcal{R} \) is also applicable, i.e., \( \mathcal{R} \) it does not miss any sites where a concretization can be affected by \( R \). By Condition 1 of Definition 5.2, if there is a NAC in \( \mathcal{N} \) that has no \textit{Maybe} elements then \( \mathcal{R} \) does not apply at \( K \). But a NAC without \textit{Maybe} in the base graph of \( M \) means that this NAC appears in every concretization of \( M \) and thus the classical rule \( R \) does not apply to any concretization either and thus applying the lifted rule does not miss any classical rule applications.

Condition 2 says that \( \Phi_M \land \Phi_{\text{apply}} \) must be satisfiable for \( \mathcal{R} \) to apply, which happens iff there exists a concretization of \( M \) where \( C^r \) and \( D^r \) are present and no NAC in \( \mathcal{N} \) is present – exactly the classical applicability condition in Definition 2.8. If this condition does not hold, there are no classical rule applications in any concretization; therefore, the lifted rule applicability condition is correct.

We now argue that the lifted rule application in Definition 5.3 is correct. To do this, we show that if \( \mathcal{R} \) satisfies the applicability conditions, then applying \( \mathcal{R} \) at a site \( K \) has the same effect as applying \( R \) at \( K \) in each concretization. Case 1 says that when \( K \) contains no \textit{Maybe} elements, we apply the rule classically to the base graph of \( M \). Without \textit{Maybe} elements, \( K \) occurs in every concretization of \( M \) and so the classical application of \( R \) in every concretization would be identical to applying \( R \).

Case 2 applies when \( K \) has some \textit{Maybe} elements. In this case, the concretizations are split into those where \( R \) does not apply and those where it does. We then aim to show that the steps (a-d) for constructing the graphical and propositional effect of applying \( \mathcal{R} \) are “correct by construction”. We do not repeat this argument, described in Section 5.2.2, here, for brevity. Thus, we conclude that the lifted rule application is also correct. Since both the applicability condition and the effect of application are correct, we conclude that \( \mathcal{R} \) satisfies the specification of correct lifting in Definition 5.1.

5.3 Properties of Lifted Transformations

In this section, we show that lifting preserves the confluence and termination of graph rewriting systems (GRSs) (see Section 2.3.2), i.e., that if a GRS has some these properties, its version containing lifted transformations does as well.

5.3.1 Termination

We need to prove the following theorem:

**Theorem 5.2** (Preservation of termination). Assume a partial model \( M \), a set \( S \) of graph rewriting rules \( \{R_1, \ldots, R_n\} \) and the set \( S = \{R_1, \ldots, R_n\} \) containing lifted version of the rules in \( S \). If for every concretization \( m_i \in \mathcal{M} \) the GRS \( G_i = \langle S, m_i \rangle \) is terminating, then the GRS \( \mathcal{G} = \langle S, M \rangle \) is also terminating.

Without loss of generality, we restrict ourselves to a rule set containing a single classical rule \( R \) which we assume is terminating. Since \( \mathcal{R} \) is correct according to Definition 5.1, repeatedly applying it to a partial model \( M \) has the same effect as repeatedly applying \( R \) to each concretization of \( M \). Since \( R \) is terminating for every concretization in \( M \), it eventually is no longer applicable to any of them. At this point, \( \Phi_{\text{apply}} \), which encodes classical applicability, is False and thus \( \Phi_M \land \Phi_{\text{apply}} \) is not satisfiable, and, by Condition 2 of Definition 5.2, \( \mathcal{R} \) does not apply. Thus, when the application of \( R \) terminates, the
application of \( R \) terminates as well. Therefore, if \( R \) is terminating for each concretization, so is \( R \) for the partial model encoding them.

### 5.3.2 Confluence

We argue that if a set of classical rules is confluent then the corresponding set of lifted rules is also confluent “up to an equivalence”, that is, when the process terminates, the resulting partial model has the same set of concretizations, regardless of the order in which the rules have been applied. We therefore need to prove the following theorem:

**Theorem 5.3 (Preservation of confluence).** Assume a partial model \( M \), a set \( S \) of graph rewriting rules \( \{ R_1, \ldots, R_n \} \) and the set \( S = \{ R_1, \ldots, R_n \} \) containing lifted version of the rules in \( S \). If for every concretization \( m_i \in [M] \) the GRS \( G_i = \langle S, m_i \rangle \) is confluent and terminating, then the GRS \( G = \langle S, M \rangle \) is also confluent.

Since for every concretization \( m_i \in [M] \) the GRS \( G_i = \langle S, m_i \rangle \) is confluent and terminating, for each one there exists an “ultimate” transformed version \( h_i \) such that \( m_i \xrightarrow{S}^* h_i \). Since \( R \) is correct according to Definition 5.1, repeatedly applying lifted rules to a partial model \( M \) has the same effect as repeatedly applying the corresponding classical rules to each concretization of \( M \). In other words there is a partial model \( H \) such that \( M \xrightarrow{S}^* H \), where \([H] = \{ h_i | m_i \xrightarrow{S}^* h_i, m_i \in [M] \}\). Thus, the lifted rule set is confluent “up to an equivalence”.

### 5.4 Evaluation

We applied our lifting approach to the problem of mapping simple UML class diagrams to relational database schemas. This problem is called “Object-Relational Mapping” (ORM) and is often used as a benchmark for model transformations [Bézivin et al., 2006]. Our aim was to gather evidence about how the lifting approach scales as uncertainty increases. We thus measured the runtime of performing ORM with lifted rules while increasing levels of uncertainty and compared it with the baseline runtime of performing ORM for a classical model. The ORM transformation we used was published in [Varró et al., 2006] and consists of 5 layered transformation rules that, given a class diagram, create a relational schema and traceability links. The rules and the source and target metamodels are shown in Appendix C.

We used the class diagram specification of the Ecore metamodel [Steinberg et al., 2009] as input to the ORM rules. Serializing Ecore models in a database is an important technical problem that has resulted in the establishment of two Eclipse projects, CDO [Eclipse, accessed 2013-03-16a] and Teneo [Eclipse, accessed 2013-03-16b], both of which implement ORM for Ecore. We manually flattened the Ecore metamodel and adapted it to the type graph used by the ORM rules in [Varró et al., 2006]. The resulting model consisted of 65 model elements: 17 classes, 17 associations, 6 generalization links and 25 attributes. Starting with a partial model with a single concretization (no uncertainty), we gradually increased the degree of uncertainty by adding more concretizations, by a step of roughly 50, thus creating models with 1, 24, 48, 108, 144, 192, and 256 concretizations. To accomplish that we incrementally injected points of uncertainty, annotating elements with Maybe and creating the corresponding May formulas. The most uncertain case (256 concretizations) contained 8 points of uncertainty, expressed across a total of 14 Maybe elements. We show the partial Ecore metamodel with the largest set of concretizations in Appendix C.
We implemented the lifting of the ORM rules using Henshin [Arendt et al., 2010]. For the satisfiability check required in Definition 5.2, we used the Z3 SMT solver [De Moura and Bjørner, 2011]. We used MMINT [Di Sandro et al., 2015] as the integration platform. We executed the case study on a computer with Intel Core i7-2600 3.40GHz × 4 cores (8 logical) and 8GB RAM, running Ubuntu-64 12.10. We applied the set of lifted rules to each input partial model and recorded the total runtime and the size of the resulting May formula. Our observations are shown in Table 5.1.

The results show that the total runtime remains almost constant at roughly 32.8 seconds, except for the largest category where it increases to 48.4 seconds. On the other hand, we see a dramatic increase in the size of the May formula, from 27.9 KB for the smallest category, to approximately 320.6 MB for the largest. This exponential growth in size is expected, given that the output May formula is constructed using existential quantification (Definition 5.3). Overall, the results suggest that lifting scales reasonably with respect to time, whereas the increasing size of the May formula may be a problem. However, we note that our implementation did not attempt to incorporate any formula simplification heuristics, and therefore there is room for optimization.

### 5.5 Related Work

Partial models encode a set of classical models, and lifted rules are rules that can transform entire sets of models simultaneously. In the following, we discuss work related to transformations that apply to modeling formalisms that represent sets of models.

Different variants of feature models have been proposed in the literature to encode a set of possible configurations of a software product line [Schobbens et al., 2006]. Transformations of feature models, i.e., the creation of a feature model representing a subset of the original, have been studied in [Czarnecki et al., 2004] under the name of feature model specialization. This process is described as a series of operations such as “feature cloning” and “reference unfolding” and resembles the uncertainty-reducing refinement described in Chapter 6. Graph transformations have also been applied to feature models, e.g., in [Segura et al., 2008], they are used to refactor product lines via feature model merging.

Transformations that apply to metamodel definitions also transform sets of models, i.e., the set of possible instances of the metamodel. The Object-to-Relational Mapping transformation [Varró et al., 2006] in Section 5.4 is one such example. Similarly, special purpose transformation languages have been built to transform ontologies, such as a rule based language based on xOWL [Wouters and Gervais, 2012].

The main difference between these transformations and the lifting approach presented here is that they are tailored to specific tasks, whereas lifting applies to arbitrary transformation rules. Moreover, these techniques only indirectly affect the classical models (e.g., variants or instances) represented by the abstraction formalism. On the other hand, lifted transformations match and transform the alternatives directly, via the propositional part of the lifted rules.

<table>
<thead>
<tr>
<th>Number of concretizations:</th>
<th>1</th>
<th>24</th>
<th>48</th>
<th>108</th>
<th>144</th>
<th>192</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Maybe elements:</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Time (sec):</td>
<td>32.6</td>
<td>32.8</td>
<td>32.7</td>
<td>32.9</td>
<td>32.6</td>
<td>33.0</td>
<td>48.4</td>
</tr>
<tr>
<td>Size of May formula (KiB):</td>
<td>0</td>
<td>27.9</td>
<td>14.0</td>
<td>1,080.9</td>
<td>1,153.4</td>
<td>19,361.9</td>
<td>320,570.7</td>
</tr>
</tbody>
</table>
We have adapted the lifting of graph-based transformations so that it can be applied to annotative software product lines [Salay et al., 2014]. An annotative product line consists of a domain model, a feature model, and a feature mapping, that annotates elements of the domain model with presence conditions, i.e., propositional expressions over features (see also Section 3.5.3) Applying lifted transformations to product lines involves transforming the domain model and then manipulating the presence conditions such that the resulting product line satisfies a correctness criterion very similar to the one in Definition 5.1. Lifting transformations for product lines is more efficient than for partial models: (a) propositional manipulations deal only with the presence conditions of elements at the transformation application site, and (b) presence conditions tend to be simpler expressions than a full blown May formula. We have taken advantage of this fact to create support for transforming industrial-grade software product lines [Famelis et al., 2015b] by lifting DSLTrans [Selim et al., 2014; Lúcio et al., 2010], a model transformation language that specializes in helping developers create provably correct transformations.

5.6 Summary

In this chapter, we have shown how to apply graph-based model transformations to partial models. We have introduced the concept of lifting: applying a transformation to a partial model should produce a new partial model which should be the same as if we had transformed each concretization of the input partial model separately and then merged them back. This means that existing transformations remain unchanged: it is the semantics of the transformation engine that changes. Specifically, we defined a three-step process for applying lifted transformations: given a matching site, first we use a SAT solver to determine whether the rule is applicable to at least one concretization at that site; then we transform the graphical part of the partial model; finally we transform the May formula to ensure that the correctness of lifting is respected. We showed that our process preserves lifting correctness, as well as confluence and termination.

We have applied our approach to the Object-Relational Mapping, a common benchmark for model transformation techniques. Specifically, we systematically injected uncertainty to the Ecore metamodel, resulting in partial models with increasing degrees of uncertainty. The benchmark indicates that there is a trade-off between low runtime and the growth of the May formula.

We discuss how the lifting of transformations fits within the overall context of Model-Driven Engineering methodology in Chapter 7.
Chapter 6
Removing and Resolving Uncertainty

If developers lack information required to select among multiple possibilities, they can use partial models to explicate their uncertainty within their software artifacts. As discussed in Section 3.1, an important characteristic of partial models is that the size of the set of concretizations reflects the modeller’s degree of uncertainty. A larger set of concretizations corresponds to a higher degree of uncertainty. Using partial model reasoning and lifted transformations developers can defer premature resolution of uncertainty while still being able to perform engineering tasks. This means that, while decisions are deferred, the size of the set of concretizations remains unchanged (cf. the definition of lifting in Section 5.2). In this chapter, we discuss how developers can resolve uncertainty once deferral is no longer necessary.

The topic of decision making in the presence of uncertainty is extensively studied in diverse fields of research such as Decision Theory, Design Space Exploration, and others. We limit our focus to the question of leveraging partial models to support the resolution of uncertainty. In other words, our aim here is not to help developers make decisions. Rather, once decisions are made (using whatever rationale), our goal is to correctly incorporate this new information to partial models.

We formally introduced refinement as a relationship between partial models in Definition 3.5: “Given two partial models \(M_1\) and \(M_2\), we say that \(M_2\) refines \(M_1\) (or that \(M_1\) is more abstract than \(M_2\)), denoted \(M_2 \preceq M_1\), iff \(C(M_2) \subseteq C(M_1)\), and \(M_2\) is consistent.” In this chapter, we operationalize this definition, and describe refinement as an engineering task that reduces the number of concretizations of a partial model, either manually or automatically.

The rest of this chapter is organized as follows: In Section 6.1, we describe how to refine partial models, based on the formal semantics of refinement introduced in Section 3.1. Specifically, we introduce two ways of accomplishing this: manual decision making and property-driven refinement. In Section 6.2, we study experimentally the efficiency and sensitivity of the property-driven refinement technique. In Section 6.3, we discuss related work, and we conclude in Section 6.4. Proofs of theorems are found in Appendix A.2.

The contents of this chapter, with the exception of Section 6.1.1, have been published in [Famelis et al., 2012] and expanded in the manuscript [Famelis et al., 2015c], currently under review.
6.1 Uncertainty-Removing Refinement

In this section, we discuss how a modeller can incorporate new information into a partial model, thus resolving uncertainty. Since the degree of uncertainty in a partial model is determined by the size of its set of concretizations, reducing uncertainty is tantamount to reducing the size of this set. If this is done incrementally we refer to this process as removal of uncertainty. Alternatively, it can be done at once by reducing the set of concretizations to a single solution, in which case we say that uncertainty is resolved. The theory of partial models allows accomplishing both tasks in a well defined, formal process of refinement [Salay et al., 2012c]. As discussed in Section 4.1.1, partial model refinement is structure-preserving as opposed to trace-preserving.

In the following, we describe the pragmatics of refinement, describing two ways to accomplish it. The first way, which we call “manual decision making”, involves point-wise modifications to the annotations of partial model elements. The second, which we call “property-driven refinement”, entails using properties to constrain the set of concretizations. Manual decision making and property-driven refinement are analogous to the Graphical and Propositional Reduced Forms (GRF and PRF), introduced in Section 3.3. On the one hand, like GRF, manual decision making puts the focus on working with the diagrammatic aspect of partial models. On the other hand, like PRF, property-driven refinement puts the focus on working with their logical aspect.

6.1.1 Manual decision making

A partial model can be refined by manually changing the annotations of its atoms and the May formula to decrease the level of uncertainty. These changes reflect the modeller’s judgement about how the newly acquired information should be represented in the model.

Performing manual decision-making is accomplished by using two atomic partial model operators, named Keep and Drop.

Invoking Keep for a partial model element amounts to making a decision that the element should be present in the model.

**Definition 6.1 (Keep).** Given a partial model $M_1 = \langle G_1, ann_1, \phi_1 \rangle$ and an element $e \in G_1$ such that $ann_1(e) = Maybe$, the operator Keep generates a new partial model $M_2 = \langle G_2, ann_2, \phi_2 \rangle$ such that:

- $G_2 = G_1$;
- $ann_2(e) = True$ and $\forall a \in G_2, a \neq e, ann_2(a) = ann_1(a)$, i.e., change the annotation of $e$ to True;
- $\phi_2 = \phi_1[True/atomToProposition(e)]$, i.e., replace $e$’s encoding by True

The function $atomToProposition()$ is defined in Section 2.1.2.

Conversely, invoking Drop for a partial model element amounts to deciding that it should not be part of the model. To ensure that Drop always results in a well-formed diagram, we define two variants, one for elements that are edges and one for vertices.

**Definition 6.2 (DropEdge).** Given a partial model $M_1 = \langle G_1, ann_1, \phi_1 \rangle$ and an edge element $e \in G_1$ such that $ann_1(e) = Maybe$, the operator DropEdge generates a new partial model $M_2 = \langle G_2, ann_2, \phi_2 \rangle$ such that:

---

1 We do not imply that either refinement method is better suited to a particular reduced form.
• \( e \notin G_2 \) and \( \forall a \neq e : a \in G_2 \Leftrightarrow a \in G_1 \), i.e., delete \( e \);

• \( \forall a \in G_2, \text{ann}_2(a) = \text{ann}_1(a) \);

• \( \phi_2 = \phi_1[\text{False} / \text{atomToProposition}(c)] \), i.e., replace \( e \)'s encoding by \( \text{False} \)

**Definition 6.3 (DropVertex).** Given a partial model \( M_1 = (G_1, \text{ann}_1, \phi_1) \) and a vertex element \( e \in G_1 \) such that \( \text{ann}_1(e) = \text{Maybe} \), and there are no incoming or outgoing transitions to \( e \), the operator DropVertex generates a new partial model \( M_2 = (G_2, \text{ann}_2, \phi_2) \) such that:

• \( e \notin G_2 \) and \( \forall a \neq e : a \in G_2 \Leftrightarrow a \in G_1 \), i.e., delete \( e \);

• \( \forall a \in G_2, \text{ann}_2(a) = \text{ann}_1(a) \);

• \( \phi_2 = \phi_1[\text{False} / \text{atomToProposition}(c)] \), i.e., replace \( e \)'s encoding by \( \text{False} \)

DropVertex has a strong requirement that only Maybe vertices without incoming or outgoing edges can be removed from the model. This has two implications: First, Maybe edges connected to the vertex should be removed first, using DropEdge. Second, if a vertex is connected with edges annotated with True, then it cannot be removed from the model via refinement. This is because annotating an edge with True indicates that the modeller is certain that it should be in the model; thus, removing it to satisfy graph well-formedness would be counter-intuitive. Instead, the strict requirement ensures that this apparent contradiction is scrutinized, potentially exposing faults in the original modelling of uncertainty.

Keep, DropEdge and DropVertex are atomic (in the sense that they affect one element at a time) refining operators:

**Theorem 6.1.** Given a partial model \( M \), and an element \( e \in M \), \( \text{Keep}(M,e) \preceq M \) and \( \text{DropEdge}(M,e) \preceq M \) if \( e \) is an edge or \( \text{DropVertex}(M,e) \preceq M \) if it is a vertex.

Resolving uncertainty about a decision point is therefore a matter of repeatedly applying Keep, DropEdge and DropVertex for all Maybe elements that are relevant to it. For example, the partial model \( M_4 \) in Figure 1.6 can be refined to the one in Figure 1.7 through a series of invocations of the operators, such as the series: \( \text{Keep}(M_4, \text{At}) \); \( \text{Keep}(M_4, \text{H2}) \); \( \text{Keep}(M_4, \text{Et}) \); \( \text{Keep}(M_4, \text{Fs}) \); \( \text{Keep}(M_4, \text{H4}) \); \( \text{Keep}(M_4, \text{Ga}) \); \( \text{Keep}(M_4, \text{Gt}) \); \( \text{DropEdge}(M_4, \text{Aa}) \); \( \text{DropEdge}(M_4, \text{Ba}) \); \( \text{DropEdge}(M_4, \text{Bt}) \); \( \text{DropEdge}(M_4, \text{Da}) \); \( \text{DropEdge}(M_4, \text{Dt}) \); \( \text{DropEdge}(M_4, \text{Ea}) \); \( \text{DropVertex}(M_4, \text{H1}) \). No action is taken for the Maybe elements \( \text{Ca}, \text{Ct}, \text{H3}, \text{H5} \). The May formula shown in Figure 1.7 has been simplified.

We note that this method of making decisions also implies that: (a) no new elements can be added to the partial model, (b) elements that are annotated with True cannot be deleted, and (c) the annotations of True elements cannot be changed. Applying one of the two operators to each Maybe element of the partial model, obviously results in a concretization (cf. Definition 3.2).

Decision-making can be automated, by creating batch uncertainty-reducing transformations (URT). A simple example of an URT would be one that deletes all Maybe atoms. URTs are different from the lifted transformations discussed in Chapter 5. Lifting aims to apply to partial models transformations written for non-partial models (such as refactoring, translations, etc.) and aims at preserving rather than reducing the set of concretizations. The main concern with using URTs is ensuring that they are indeed refining. We have studied this for URTs based on graph-rewriting, applied to a subclass of partial models that do not have May formulas in [Salay et al., 2015]. For such partial models, proving that a URT is refining requires proving that the rule’s right hand side (RHS) is a refinement of its left hand side (LHS).
Chapter 6. Removing and Resolving Uncertainty

Algorithm 5: Computing all concretizations of a given model that satisfy a given property.

Input: A partial model $M = \langle G_M, \text{ann}_M, \phi_M \rangle$ of type $T$ and a grounded property formula $\phi_p$

Output: A partial model $M^p \preceq M$, such that $\forall m \in C(M^p) \cdot m \models \phi_p$

1. Generate $M^{PRF}$ from $M$ using Algorithm 2;
2. $G_r := G_M^{PRF}$;
3. foreach atom $a \in G_M^{PRF}$ do $\text{ann}_r(a) = \text{Maybe}$;
4. $\phi_r := \phi_M^{PRF} \land \phi_p$;
5. $M_r := \langle G_r, \text{ann}_r, \phi_r \rangle$;
6. Generate $M^p := \text{GRF}$ of $M_r$ using Algorithm 1;
7. return $M^p$;

6.1.2 Property-driven refinement

Property-driven refinement provides a means to resolve uncertainty in a declarative way. Instead of manually making decisions about each Maybe element separately, the modeller specifies her refinement rationale using a property. Using the reasoning technique described in Chapter 4, the modeller is able to determine whether the property holds for all, some or none of the partial model’s concretizations. The partial model property checking decision procedure Section 4.1.2 returns True, Maybe, and False respectively. If the result of property checking is True or False, the model already satisfies or violates the property regardless of how uncertainty is resolved. This may signify that the specified property is not a good guide for uncertainty resolution, that there is nothing the developer needs to do, or that there is nothing the developer can do without first changing the partial model. In this section, we focus on the third possibility, i.e., when the result of property checking is Maybe.

If the result of property checking is Maybe, the developer can use the property to constrain the partial model so that it ends up with a subset of the original set of concretizations. Effectively, the property can be used as a “filter” to remove unwanted possibilities from the partial model and ensure that all of its concretizations satisfy the property. In the PtPP example, assume that the team decides that the property P4 (“Users can always cancel any operation, i.e., every non-idle state has a transition to Idle on cancel”) should be satisfied by the model $M_{p2p}$ shown in Figure 1.3. The team should therefore be able to restrict their working set of alternatives to those that satisfy it, i.e., those in Figure 1.4(a-d).

Assume that the result of verification of an important property $p$ on a partial model $M$ of type $T$ is Maybe. The subset of concretizations of $M$ that satisfy $p$ is exactly characterized by the formula $F^+(T, M, p)$ defined in Section 4.1.2. We use this formula to construct the partial model $M^p$ using Algorithm 5. In particular, first we put $M$ into PRF (Line 1). Then, we construct an intermediate partial model $M_r$ (Lines 2-5) directly in PRF, by copying over the graph of $M$, annotating all atoms with Maybe and setting its May formula to $\phi_M^{PRF} \land \phi_p$. This way, we add the property as part of the May formula and thus restrict the set of concretizations to only those satisfying the property. Finally, we put $M_r$ into GRF, using Algorithm 1, and return it (Lines 6-7).

Since the conversion of $M$ to PRF and the creation of the intermediate model $M_r$ are linear in the number of atoms of the input model, the complexity of the algorithm is that of the GRF conversion, i.e., NP-hard.

Using Algorithm 5, we can refine a partial model using a grounded property as a filter for its concretizations:

Theorem 6.2. Let model $M^p$ be a result of running Algorithm 5 on a model $M$. Then, $M^p \preceq M$ and
\[ \forall m \in C(M^p) \cdot m \models \phi. \]

In the PtPP example, consider the property \( P5 = \neg P2 \), where \( P2 \) is the property "no two transitions have the same source and target". The concretizations of \( M_{p2p} \) that satisfy \( P5 \) are those that satisfy the formula \( F^+(SM, M_{p2p}, P5) \), i.e., those in Figure 1.4(c, d). The partial model \( M_{p2p}^{P5} \) that represents them is shown in Figure 3.4(a). To construct it, we first put the model \( M_{p2p} \) in Figure 1.3 in PRF. We then conjunct its May formula with \( \phi_{P5} \). After removing all superfluous atoms and simplifying, we end up with the model in Figure 3.4(b). Creating its GRF equivalent results in the model in Figure 3.4(a).

Note that in the example above, since \( P5 = \neg P2 \), we used property-driven refinement to create a partial model that only has concretizations that violate the property \( P2 \). As discussed in Section 4.2.3, this is a form of feedback, called diagnostic core, that can be generated after property checking to help users understand the commonalities between concretizations that violate a property.

### 6.2 Experimental Study of Property-driven Refinement

In this section, we present an experimental evaluation of property-driven refinement. Contrary to the experimentation presented in Section 4.3, it is not meaningful to compare property-driven refinement with some "classical" approach. Specifically, while it is possible to compute the set of all possible concretizations that satisfy a property without using partial models, doing so would not have the same effect as performing property-driven refinement. The reason is that property-driven refinement creates a GRF partial model which is a visual representation of the set.

This allows interacting with this set in ways that are not possible or meaningful with an enumerated set of classical models. For example, in PtPP, when reasoning with the property \( P2 \) ("no two transitions have the same source and target"), generating a diagnostic GRF partial model, such as \( M^{\neg P2} \) in Figure 3.4(a), makes it easy to immediately visualize what elements are shared among all concretizations that violate the property (e.g., the transition from Leeching to Idle on completed). A "classical" approach would require developers to go through each model in the set of counter-examples (here, the models in Figures 1.4(a,b,e,f)) to notice that it is shared amongst all of them.

Therefore, the objective of this experimental evaluation is studying alternative ways for implementing property-driven refinement. We describe them below.

#### 6.2.1 Experimental setup

Accomplishing property-driven refinement requires the use of Algorithm 5. At the last step of the algorithm (Line 6), we create a GRF equivalent of an intermediate model \( M_r \) using the GRF construction (Algorithm 1). However, in Section 3.4, we showed (Theorem 3.6) that Algorithm 3 can also be used as a "brute-force" algorithm to compute the GRF of a partial model. This can be done by first breaking the partial model down to its constituent concretizations and then merging them again, using Algorithm 3.

We conducted an empirical study to compare these two alternative methods for creating GRF diagnostic partial models. In particular, we attempted to answer the following research questions:

**RQ3**: Which alternative method is more efficient for creating GRF models?

**RQ4**: How sensitive are the two alternative methods to the varying model sizes and degree of uncertainty?
To get answers to RQ3 and RQ4, we set up experiments as in Section 4.3: E3 and E4. In experiment E3, we measured the performance of computing the GRF using Algorithm 1, as described in Section 4.2.3. In experiment E4, we measured the performance of the alternative method that uses Algorithm 3 as described above.

To answer RQ3, we studied the results of the two experiments and compared the observed runtimes. To answer RQ4, we executed the experiments E3 and E4 with randomly generated experimental inputs as in Section 4.3.

### 6.2.2 Inputs, implementation and methodology

To run experiments E3 and E4, we used the same parameters and parameter values as the ones described in Section 4.3.2 and reused the implementation described in Section 4.3.3.

Executing the experiment E3 requires us to calculate the backbone of a formula. We implemented the backbone computation on top of the Z3 SMT solver [De Moura and Bjørner, 2011] using incremental solving and several obvious optimizations, natively supported by Z3. For experiment E4, we used incremental solving to collect the set of all concretizations of the input model. We integrated these in the experimental driver described in Section 4.3.4.

Property-driven refinement is only meaningful in the case where a property check returns `Maybe`, as discussed in Section 6.1.2. Given a combination of parameters (cf. Tables 4.2, 4.3 and 4.4), the experimental driver generates an input as described in Section 4.3.4 and then performs a property check, as described in Section 4.1.2. If the result is not `Maybe`, the driver skips this input.

We ran the experiments on the same machine and with the same configuration as in Section 4.3.4. For each run, we recorded the observed run-times and calculated the speedups $S_{3/4} = \frac{T_{E3}}{T_{E4}}$ and $S_{4/3} = \frac{T_{E4}}{T_{E3}}$, where $T_{E3}$ was the time to compute the GRF based on Algorithm 1 (experiment E3) and $T_{E4}$ the time to compute it based on Algorithm 3 (experiment E4).

### 6.2.3 Results

The experiments did not show dramatic differences in speedup between the different properties, whereas we observed large differences in speedup stemming from differences in model size and size of set of concretizations. We observed the largest $S_{4/3}$ speedup for S-sized models with XL-sized sets of concretizations (7.16) and the smallest for XL-sized models with S-sized sets of concretizations (0.01). These correspond to the smallest $S_{3/4}$ speedup at 0.14 and the largest at 143.40, respectively. For models with bigger sets of concretizations, the highest $S_{3/4}$ speedup was observed for XL-sized models with M-set of concretizations (4.97).

We plot both speedups to highlight the advantages and disadvantages of each method. The ranges of speedups $S_{1/3}$ and $S_{3/4}$ are shown in Figure 6.1(a) and Figure 6.1(b), respectively. The plotted values are averages over all properties for each combination of model size and size of set of concretizations. Some clear trends arise from these observations: The conversion to GRF based on Algorithm 1 performs better for partial models of smaller sizes that have larger sets of concretizations. The alternative approach (finding all concretizations and then constructing a new partial model using Algorithm 3) works best for models with small sets of concretizations. The observed speedups for very large models with small sets of concretizations were dramatic.

Based on the summarized observations, we conclude, regarding RQ3 (feasibility), that when it comes
to selecting a method for generating diagnostic feedback, there exists a clear trade-off between model size and size of the set of concretizations. Specifically, for smaller models with larger sets of concretizations, Algorithm 3 should be used, while for larger models with smaller sets of concretizations — Algorithm 1.

This points us to an answer regarding RQ4 (sensitivity). The observations in Figure 6.1 lead to the conclusion that the two methods are very sensitive to varying model sizes and degree of uncertainty. Specifically, the speedup offered by each method is dependent on whether the approach is applied to partial models in its niche regarding model size and set of concretizations.

Because of its design, the experimental comparison presented in this section suffers from the same threats to validity as the ones discussed in Section 4.3.6.
Chapter 6. Removing and Resolving Uncertainty

6.3 Related Work

**Refinement** In software engineering, refinement generally means an operation that elaborates some artifact (e.g., a program, a model, a specification) [Wirth, 1971]. In essence, given an artifact, refinement creates a new one that is less abstract. In the literature, we encounter two notions of refinement that are contradicting albeit subtly different.

On the one hand, in the context of formal methods and, specifically, software specification, refinement is understood as increasing the requirements of a specification. For example, in refinement calculi [Morgan, 1990], refinement can be accomplished by strengthening a specification’s post-condition or weakening its pre-condition. In the context of partial behavioural formalisms, such as Modal Transition Systems (MTSs), refinement entails creating a new MTS with a subset of the traces of the original [Larsen and Thomsen, 1988]. In both cases, a refined specification admits fewer implementations that the original. In that sense, refinement is understood as *removal* of behaviour.

On the other hand, in the context of model-driven engineering, refinement is understood as a process by which implementation details are added to a model expressed at a high level of abstraction [Sendall and Kozaczynski, 2003]. This is typically done using vertical model transformations [Mens and Gorp, 2006], such as code generation. Refinement is thus *addition* of (low level) behaviour. Model Driven Architecture (MDA) relies heavily on this notion of refinement; however, it also encompasses the “removing” notion of refinement for “internal refinements”, i.e., transformations that create models at the same level of abstractions, while preserving semantics [Paige et al., 2005].

In the context of partial modelling, refinement, both in its manual and its property-driven flavours, is understood as an operation that *removes uncertainty*, without changing levels of abstraction. In that sense it is consistent with MDA’s “internal refinements”, albeit addressing a different concern. However, as discussed in detail in Section 4.1.1, an important difference between partial model refinement and other forms of “removing” refinement, such as behavioural model refinement, is that its focus is on preserving structure, rather than behaviour (traces).

**Set-reducing operations.** Uncertainty resolution is an example of an operation that, given an artifact encoding a set of models, produces a new one that encodes a subset of the original. As discussed in Section 3.5, such artifacts are found in formalisms such as software product line engineering (SPLE), megamodelling and metamodelling.

In SPLE, staged configuration [Czarnecki et al., 2004] is a method for incrementally making choices about which features to include in a product. This is achieved by stepwise reducing the set of possible configurations of the input feature model. This is done systematically using six different specialization steps, each one implementing a way to remove a configuration choice. These steps are formalized using context-free grammars [Czarnecki et al., 2005] and are implemented in the FeaturePlugin tool for the Eclipse platform [Antkiewicz and Czarnecki, 2004].

Metamodels are also understood as abstractions of sets of instance models. A method typically used to reduce the set of instances of a metamodel is the addition of constraints [Karsai et al., 2000], using a language such as the Object Constraint Language (OCL) [Object Management Group, 2006b]. A different approach is to enable the creation of metamodel sub-types. There are various approaches for creating model subtypes [Guy et al., 2012], mainly focusing on achieving model substitutability, especially in the context of model transformations. Metamodel *pruning* is a dual to subtyping, aiming to create metamodel super-types [Sen et al., 2009].
Megamodels are used to model the macroscopic view of software development. The elements of a megamodel are themselves models, interconnected with various macro-relations [Favre and Nguyen, 2005; Salay et al., 2009]. In that sense, it is an abstraction of a set of models. Reducing this set (in order to, for example, create task-specific views) can be accomplished with model slicing [Blouin et al., 2011; Kagdi et al., 2005]. Since megamodels are themselves models, it is possible to use submodel extraction techniques [Carré et al., 2013].

6.4 Summary

In this chapter we described the resolution of uncertainty in partial models. A partial model is a transient artifact, created when the modeller lacks sufficient information to make informed decisions. When such information becomes available, the modeller can incorporate it in her partial model by reducing the size of its set of concretizations.

In order to create support for systematic resolution of uncertainty, we have operationalized the definition of partial model refinement given in Section 3.1 by defining two refinement approaches: manual decision making and property-driven refinement.

In manual decision making, the modeller incorporates the newly acquired information in the partial model by making decisions about whether to include or not individual partial model elements that are annotated with \textit{Maybe}. This is done using the formally defined operators \textit{Keep}, \textit{Drop} (specifically, \textit{DropVertex} and \textit{DropEdge}), respectively.

Instead of editing the diagram of a partial model using operators, property-driven refinement reduces the set of its concretizations declaratively. In this case, the modeller expresses the newly acquired information as a property. A refinement is then constructed by appropriately combining the property with the partial model’s May formula. Property-driven refinement is used in Chapter 4 for diagnosis, by creating “diagnostic cores”, i.e., refinements encoding the subset of concretizations that violate some property of interest.

The algorithm for producing user-friendly representations of refinements generated using properties requires the expensive computation of the “backbone” of the May formula. We conducted an experimental evaluation to study the feasibility and sensitivity of this method of property-driven refinement. To do this, we compared the method that uses backbone computation to a brute-force method that enumerates all concretizations of interest and then uses Algorithm 3 to encode them in a partial model. We have found that the different approaches work best for different combinations of model size and size of the set of concretizations.

In Chapter 7, we introduce a methodological approach that contextualizes uncertainty resolution in the software lifecycle with respect to the rest of partial model operations.
Chapter 7

A Methodology for Uncertainty Management

In this chapter, we investigate the implications of uncertainty management with partial models on software engineering methodology. We outline the lifecycle of uncertainty management, identifying its different phases from modelling to resolution. This allows us to contextualize each of the partial modelling operations defined in previous chapters and so to put together a coherent methodological framework for uncertainty management. We illustrate our approach using the Model Driven Architecture (MDA) as a non-trivial example of adapting a software engineering methodology to incorporate uncertainty management. Finally, we present Mu-MMINT, an interactive modelling tool for uncertainty management in software development.

The rest of the chapter is organized as follows. In Section 7.1, we introduce the “Uncertainty Waterfall”, an abstract timeline of uncertainty in software artifacts. In Section 7.2, we use the phases of the Uncertainty Waterfall to describe the usage context of each partial modelling operation. In Section 7.3, we exemplify this approach by applying it to the Model Driven Architecture (MDA) software methodology. In Section 7.4, we present Mu-MMINT, an Eclipse-based tool for managing partial models. We present related work in Section 7.5 and conclude in Section 7.6.

7.1 The Uncertainty Waterfall

The main premise of this thesis is that once uncertainty appears in the development process, a developer should attempt to capture it in a special kind of software artifact, called a partial model. Using partial models, the developer can continue working while avoiding decisions that, in the absence of enough information, would be premature. When sufficient information is available, the developer systematically refines the partial model, such that the new information resolves the encoded uncertainty.

In this section, we introduce an abstract model for capturing this lifecycle, that is intrinsic to uncertainty management. This model is called “Uncertainty Waterfall” and presents an idealized timeline of partial model use. It is shown in Figure 7.1 and it consists of three stages: Articulation of uncertainty, Deferral of decisions, and Resolution of uncertainty.

We discuss each one in more detail below.
Chapter 7. A Methodology for Uncertainty Management

Figure 7.1: The Uncertainty Waterfall: an idealized timeline of uncertainty management.

Articulation stage:
This stage is the entry point of uncertainty management: a developer is uncertain about some aspect of her model due to insufficient information. The main goal during this stage is therefore to change the model, so that it reflects the developer’s uncertainty. During this stage, the degree of uncertainty in the model increases as the developer encodes in a partial model the different possible ways to resolve her uncertainty. These are encoded as concretizations of the partial model.

Deferral stage:
In this stage, the main goal is to avoid premature decision making, while still being able to make use of software engineering techniques. To do that, the developer uses the partial model as the primary development artifact. The developer must therefore use versions of such techniques that have been appropriately lifted so that they are applicable to partial models. During this stage the degree of uncertainty in the model remains unchanged. This is a fundamental property that lifted tasks must preserve.

Resolution stage:
This stage begins when more information becomes available. The developer incorporates this new information into the partial model in a systematic way. During this stage, the degree of uncertainty in the partial model is reduced to reflect the newly acquired information. The ultimate result of this stage is a model without any partiality, i.e., a concretization of the original partial model.

The succession of these stages follows a decreasing degree of uncertainty. Uncertainty is introduced during the Articulation stage; it remains stable during the Deferral stage, and is reduced during the Resolution stage. We show this schematically in Figure 7.2, as a level of uncertainty present in the

Figure 7.2: Degree of uncertainty in different stages of the Uncertainty Waterfall.
software models plotted over time.

We have used the name “Uncertainty Waterfall” in direct analogy with the Waterfall model for software development [Sommerville, 2007]. We wish to underscore that, like the traditional Waterfall model, the Uncertainty Waterfall is an abstraction of an inherently messy process. It should not be misunderstood as a rigid prescription of a strict succession of stages, where an initial explication of uncertainty is necessarily followed by a series of partial models of monotonically decreasing uncertainty that culminates in a single concretization. Similarly, the plot in Figure 7.2 is also an abstraction. We offer the Uncertainty Waterfall model as an intuitive description of the stages of uncertainty management and the dependencies between them. In fact, the stages may overlap or be used in different orders. A more realistic model is the Extended Uncertainty Waterfall, shown in Figure 7.3, that includes backward transitions from later to earlier stages.

The Uncertainty Waterfall, shown in Figure 7.3, contains the following transitions between stages of uncertainty management:

**doWork** (Articulation→Deferral)

Once the developer has completed expressing uncertainty in her software artifacts, she can use her partial models to perform software engineering tasks. This transition is triggered once there is no more uncertainty to express and the developer needs to continue working.

**newInformation** (Deferral→Resolution)

If new information becomes available, the developer can use it to resolve all or part of her uncertainty. This transition is triggered if the developer is in a position to resolve some of the uncertainty and wishes to do so.

**moreWork** (Resolution→Deferral)

If uncertainty is only partially resolved, the developer can continue performing lifted engineering tasks while the remaining uncertainty persists. For example, the developer may perform a verification task, use the results to resolve some of the uncertainty, and then continue working with refined partial model. This transition is triggered once no more uncertainty can be resolved and the developer needs to continue working.

**newUncertainty** (Deferral→Articulation)

In the course of her work with partial models, the developer might encounter additional points of
uncertainty. For example, while already working with a partial model, she may become uncertain about some design decision for which there was previously no uncertainty. This transition is triggered when the developer faces the need to express further uncertainty in her software artifacts.

**moreUncertainty** (Resolution→Articulation)

The developer might need to articulate additional uncertainty immediately following a resolution. Similarly to the transition **newUncertainty**, this transition is triggered when the developer faces the need to express further uncertainty in her software artifacts.

**moreInformation** (Articulation→Resolution)

The developer might acquire information that allows her to resolve some uncertainty immediately following articulation. Similarly to the transition **newInformation**, this transition is triggered if the developer is in a position to resolve some of the uncertainty and wishes to do so.

In addition to these transitions, we also allow **Undo** transitions between any two stages. For example, the transition **Undo**(Resolution→Articulation) allows the developer undo some refinement in order to explore different alternatives, while the transition **Undo**(Articulation→Resolution) allows the developer to revert to a previous a more concrete version of her artifacts if she decides there is no benefit to articulating a particular point of uncertainty. Undo transitions are triggered whenever the developer is not satisfied with the immediately previous transition she took.

The backward links to the Articulation stage can also be the result of eliciting possible solutions from a previously open design decision. For example, in the PtPP scenario, the developers were initially faced with uncertainty about the design decision: *what policy is followed when leeching ends?*. This is an open-ended question and as such outside the scope of this thesis (cf. the discussion of the limits of our approach in Section 1.3). However, in our scenario the PtPP developers elicited three acceptable solutions to this design decision (“selfish”, “benevolent”, and “compromise”), thus reducing the question to uncertainty about selecting one of them. The backward transitions to the Articulation stage, therefore allow this process of gradual elicitation of specific alternatives to an open-ended decision point.

The combination of the transitions **newInformation** and **moreWork** captures the fact that resolution of uncertainty is not always immediate (i.e., producing a concrete, non-partial model in one step) but rather that the Deferral and Resolution stages can be interleaved. This interleaving represents the gradual removal of uncertainty. For example, the developer might apply a transformation to her model, then resolve some of the uncertainty, check a property, apply a second transformation, and so on.

Finally, we note that the Uncertainty Waterfall also illustrates the fundamentally transient nature of partial models: they are created when uncertainty is encountered, used as primary development artifacts in the presence of uncertainty and are ultimately discarded, collapsing to a single, concrete model.

In the following, we use the Uncertainty Waterfall to contextualize the different partial model operators described throughout the thesis. Whenever we refer to the Uncertainty Waterfall, we mean the extended version.

### 7.2 Uncertainty Management Operators

In this thesis, we propose an approach for managing uncertainty in software models by doing model management [Bernstein, 2003] of models that contain uncertainty, i.e., *partial models*. The Extended Uncertainty Waterfall allows us to identify the three distinct usage contexts (Articulation, Deferral,
Table 7.1: Operator Construct

| Description | Create a partial model from a given set of concrete models that are alternative resolutions to uncertainty. |
| Inputs | A set of non-partial models. |
| Outputs | A partial model. |
| Usage context | The developer has at her disposal a known, fully enumerated set of alternative models, but has insufficient information about which of the models is best suited for her purpose. |
| Preconditions | No partial model exists. The set of models must be known and fully enumerated. |
| Postconditions | The resulting partial model is in Graphical Reduced Form (GRF) and its set of concretizations is exactly the set of input models. |
| Limitations | The developer must have the full knowledge of the input set. |
| Implementation | The operator is realized using Algorithm 3, described in Section 3.4. |

Table 7.2: Operator MakePartial

| Description | Create a partial model from a given concrete model by introducing uncertainty to it. |
| Inputs | A concrete model and an informal description of uncertainty. |
| Outputs | A partial model. |
| Usage context | The developer uses the informal description of uncertainty and her intuition about how it should be expressed in the given model. |
| Preconditions | No partial model exists. |
| Postconditions | The input model is a concretization of the output partial model. |
| Limitations | Effectiveness depends on the intuition of the developer. |
| Implementation | The task is done manually, using the tooling described in Section 7.4. |

Resolution) in which developers work with partial models. In section, we describe the uncertainty management operators that pertain to each stage.

Each operator is described in terms of: (a) its name, (b) a high level description of its functionality, (c) its inputs and outputs, (d) a description of the development setting in which it is meaningful to invoke it, (e) its preconditions and postconditions, (f) its limitations in terms of effectiveness or usability, and (g) the section in the thesis where its implementation is described in detail.

### 7.2.1 Articulation stage

In this stage operators aim at explicating the developer’s uncertainty about some design decision. Inputs to articulation operators are therefore inherently informal and subjective. We distinguish three such operators: Construct, MakePartial, and Expand.

The operator Construct creates a partial model from a set of concrete models, as described in Table 7.1.

The Construct operator assumes that the developer has full knowledge of the set of concretizations before invoking it. However, the articulation process can also be manual, requiring the intuition of the developer to appropriately capture the space of possibilities in the partial model. This is done using the operator MakePartial, described in Table 7.2.

If uncertainty is encountered while the developer is already working with a partial model, the operator
Table 7.3: Operator **Expand**

<table>
<thead>
<tr>
<th>Description</th>
<th>Introduce additional uncertainty to a partial model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model.</td>
</tr>
<tr>
<td>Outputs</td>
<td>A partial model.</td>
</tr>
<tr>
<td>Usage context</td>
<td>The developer encounters new uncertainty during the Deferral or Resolution stages of the Extended Uncertainty Waterfall.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Some uncertainty has already been explicated in the partial model.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>The input partial model is a refinement of the output.</td>
</tr>
<tr>
<td>Limitations</td>
<td>Effectiveness depends on the intuition of the developer.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The task is done manually, using the tooling described in Section 7.4.</td>
</tr>
</tbody>
</table>

Table 7.4: Operator **Transform**

<table>
<thead>
<tr>
<th>Description</th>
<th>Apply a transformation to a partial model ensuring that all concretizations are correctly transformed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model</td>
</tr>
<tr>
<td>Outputs</td>
<td>A partial model</td>
</tr>
<tr>
<td>Usage context</td>
<td>The developer wishes each concretization to be transformed but does not wish to enumerate them all and do it for each one individually.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>The set of concretizations of the output partial model is exactly the same as if the input partial model had been broken down to its set of concretizations using the operator <strong>Deconstruct</strong>, then each concretization in that set had been transformed individually, and then a partial model had been constructed from that set using the operator <strong>Construct</strong>.</td>
</tr>
<tr>
<td>Limitations</td>
<td>The model transformation must be expressed as graph rewriting system (i.e., a set of graph rewriting transformation rules).</td>
</tr>
<tr>
<td>Implementation</td>
<td>The operator is realized by the technique described in Section 5.2.</td>
</tr>
</tbody>
</table>

**Expand**, described in Table 7.3, allows expanding the existing partial model with the new possibilities.

The common characteristic of the operators of the Articulation stage is that the size of the set of concretizations increases.

### 7.2.2 Deferral stage

The aim of operators in this stage is to facilitate decision deferral: if a developer can accomplish software engineering tasks using partial models, then there is no need to prematurely remove uncertainty. We therefore lift software engineering operators such they can be used with partial models, without affecting the degree of uncertainty. We identify two such operators: **Transform**, and **Verify**.

The operator **Transform**, described in Table 7.4, allows developers to apply a model transformation to a partial model such that all it concretizations are correctly transformed, albeit without having to enumerate them.

The operator **Verify**, described in Table 7.5, is used to determine whether a partial model satisfies a syntactic property.

Following verification, the developer may want to perform diagnostics, as described in Chapter 4. Since diagnostic operations appropriately remove uncertainty to illuminate parts of the set of the input partial model's concretizations, they are discussed in the next section.
Table 7.5: Operator Verify

<table>
<thead>
<tr>
<th>Description</th>
<th>Check whether a partial model satisfies a property.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model, a property.</td>
</tr>
<tr>
<td>Outputs</td>
<td>A value from the set { True, False, Maybe }.</td>
</tr>
<tr>
<td>Usage context</td>
<td>A developer is interested in determining whether a property is satisfied by some, all or none of the concretizations of the partial model.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>If the property is satisfied by all, none or some of the partial model’s concretizations, then the output is True, False, and Maybe respectively.</td>
</tr>
<tr>
<td>Limitations</td>
<td>The property is syntactic, i.e., its verification does not require knowledge of the semantics of the partial model’s base language.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The operator is realized by the technique described in Section 4.1.</td>
</tr>
</tbody>
</table>

Table 7.6: Operator Deconstruct

<table>
<thead>
<tr>
<th>Description</th>
<th>Produce the set of concretizations of a given partial model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model</td>
</tr>
<tr>
<td>Outputs</td>
<td>A set of concrete models</td>
</tr>
<tr>
<td>Usage context</td>
<td>The developer needs to perform a task on each concretization of the input partial model and no lifted version of the task exists.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>None.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>The output set contains exactly the set of concretizations of the input partial model.</td>
</tr>
<tr>
<td>Limitations</td>
<td>High cost.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The operator can be implemented by passing the partial model’s May formula to an All-Solutions SAT solver [Grumberg et al., 2004].</td>
</tr>
</tbody>
</table>

During the deferral stage if some software engineering operation (in addition to the ones described above) is required, then it needs to be appropriately lifted. For arbitrary operations, correct lifting generally depends on the semantics of the base language and is thus outside the scope of this thesis. It is however always possible (albeit expensive) to enumerate all concretizations, apply the non-lifted operation to each concretization separately, and then merge the results using Construct. We thus also introduce the operator Deconstruct, described in Table 7.6, that, given a partial model, creates its set of concretizations. This can be accomplished by passing the partial model’s May formula to an All-Solutions SAT solver [Grumberg et al., 2004]. All-Solutions SAT solvers are special purpose reasoning engines that specialize in efficiently computing all satisfying valuations of a boolean formula. Thus, the solver will enumerate all satisfying valuations of the May formula, which can then be translated into models, using the approach described in Section 2.1.2.

Unlike operators in the Articulation and Resolution stages, that increase and reduce the degree of uncertainty in partial models respectively, operators in the Deferral stage do not affect the degree of uncertainty.

7.2.3 Resolution stage

Operators at this stage incorporate new information to a partial model, thus reducing its degree of uncertainty. All the operators can produce either a new partial model that refines the original, or a
non-partial model, i.e., a concretization of the original. We describe two classes of operators for this stage. On the one hand, the **Decide** and **Constrain** operators incorporate new information obtained by the developer. Using them, the developer can resolve (partially or completely) the open questions that prompted the creation of partial models during the Articulation stage. On other hand, the diagnostic operators **GenerateCounterExample**, **GenerateExample**, and **GenerateDiagnosticCore** produce feedback following an invocation of the **Verify** operator. They remove uncertainty from the input partial model in order to produce subsets of its concretizations to appropriately illuminate the results of verification.

The **Decide** operator, described in Table 7.7, allows the developer to manually make decisions about which elements should and should not remain in the model. The **Constrain** operator, described in Table 7.8, allows narrowing the set of concretizations of a partial model to a subset that satisfies some property of interest.

We define three diagnostic operators. The operator **GenerateCounterExample**, described in Table 7.9 creates a concretization that functions as a witness as to why a partial model does not satisfy some property of interest. The operator **GenerateExample**, described in Table 7.10, creates a concretization that functions as a witness as to why a partial model can satisfy some property of interest, depending on how uncertainty is resolved. The operator **GenerateDiagnosticCore**, described in Table 7.11, creates a partial model that encodes the subset of concretizations of the input partial model that do not satisfy

| Description | Make decisions about whether to Keep or Drop individual Maybe elements from a partial model. |
| Inputs | A partial model and an informal description of information that resolves uncertainty. |
| Outputs | A (potentially singleton) partial model. |
| Usage context | The developer uses the informal description of newly acquired information, as well as her intuition about how it should be incorporated in the input partial model. |
| Preconditions | None. |
| Postconditions | The output partial model is a refinement of the input partial model. |
| Limitations | The developer may only use the atomic operations **Keep**, **DropEdge**, **DropVertex**. |
| Implementation | The operator is realized by the technique described in Section 6.1.1. |

| Description | Create a partial model with a subset of the concretizations of the input partial model such that all its concretizations satisfy a property of interest. |
| Inputs | A partial model, a property. |
| Outputs | A partial model. |
| Usage context | The developer has determined that concretizations of the input partial model that do not satisfy the input property are not valid ways to resolve uncertainty and should thus be excluded. |
| Preconditions | None. |
| Postconditions | The output partial model refines the input partial model. |
| Limitations | This operator is subject to the same limitations as **Verify**. |
| Implementation | The operator is realized using the technique described in Section 6.1.2. |
Table 7.9: Operator \textbf{GenerateCounterExample}

<table>
<thead>
<tr>
<th>Description</th>
<th>Create a non-partial model that illustrates why a partial model does not satisfy a property.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model, a property.</td>
</tr>
<tr>
<td>Outputs</td>
<td>A concrete model.</td>
</tr>
<tr>
<td>Usage context</td>
<td>A developer wants to diagnose why a partial model does not satisfy a property, i.e., why the result of \textbf{Verify} is not True.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>The result of the operator \textbf{Verify} using the input partial model and the input property is \textbf{Maybe}, or \textbf{False}.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>The output model is a concretization of the input partial model and satisfies the input property.</td>
</tr>
<tr>
<td>Limitations</td>
<td>This operator is subject to the same limitations as \textbf{Verify}.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The operator is realized by the technique described in Section 4.2.1.</td>
</tr>
</tbody>
</table>

Table 7.10: Operator \textbf{GenerateExample}

<table>
<thead>
<tr>
<th>Description</th>
<th>Create a non-partial model that illustrates why a partial model may satisfy a property.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model, a property.</td>
</tr>
<tr>
<td>Outputs</td>
<td>A concrete model.</td>
</tr>
<tr>
<td>Usage context</td>
<td>A developer wants to diagnose why a partial model may satisfy a property, i.e., why the result of \textbf{Verify} is not \textbf{False}.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>The result of the operator \textbf{Verify} using the input partial model and the input property is \textbf{Maybe}, or \textbf{True}.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>The output model is a concretization of the input partial model and satisfies the input property.</td>
</tr>
<tr>
<td>Limitations</td>
<td>This operator is subject to the same limitations as \textbf{Verify}.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The operator is realized by the technique described in Section 4.2.2.</td>
</tr>
</tbody>
</table>
Table 7.11: Operator GenerateDiagnosticCore

<table>
<thead>
<tr>
<th>Description</th>
<th>Create a partial model that illustrates why a partial model does not satisfy a property.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>A partial model, a property.</td>
</tr>
<tr>
<td>Outputs</td>
<td>A partial model.</td>
</tr>
<tr>
<td>Usage context</td>
<td>A developer wants to diagnose why a partial model does not satisfy a property, i.e., why the result of Verify is not True.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>The result of the operator Verify using the input partial model and the input property is Maybe, or False.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>The output partial model refines (or is equivalent to) the input partial model. The property is not satisfied by any of its concretizations.</td>
</tr>
<tr>
<td>Limitations</td>
<td>This operator is subject to the same limitations as Verify.</td>
</tr>
<tr>
<td>Implementation</td>
<td>The operator is realized by the technique described in Section 4.2.3.</td>
</tr>
</tbody>
</table>

Figure 7.4: Uncertainty management operators overlayed on the Extended Uncertainty Waterfall.

We summarize the set of uncertainty management operators by overlaying them on the Uncertainty Waterfall in Figure 7.4. Specifically, the operators Construct, MakePartial, and Expand are part of the Articulation stage, the operators Transform, Verify, and Deconstruct are part of the Deferral stage, and the operators Decide, Constrain, GenerateCounterExample, GenerateExample, and GenerateDiagnosticCore are part of the Resolution stage.

7.3 Application: Uncertainty Management in MDA

In this section, we illustrate how uncertainty management can be incorporated into existing software engineering methodologies. Specifically, we apply our approach to the Model Driven Architecture (MDA) [Object Management Group, 2014] methodology for software development. We identify the sources of uncertainty in MDA and show how the Uncertainty Waterfall can be used to weave uncertainty operators in the overall process.

As described in Section 2.4, MDA organizes software development in terms of three architectural
layers, representing hierarchical levels of abstraction. Models at the top layer model the business domain and are called “computation independent models” (CIMs). In each layer, developers create models that are independent of lower-level details. Models at lower levels of abstraction are derived automatically using transformations. These “vertical” transformations combine upper-layer “platform independent models” (PIMs) with descriptions of the “platform” of the target layer, to create “platform specific models” (PSMs). Transformations are also used “horizontally”, i.e., within a single layer, to produce derivative representations of PIMs.

In the following, we list the main sources of uncertainty in MDA and describe how the operators of the Uncertainty Waterfall can be used to manage it. Specifically, in MDA, uncertainty can arise:

(a) while creating the PIM models of each layer,

(b) from insights gained from derivative representations of PIMs,

(c) in the description of platforms,

(d) in the choice of platform, and

(e) in transformation models.

Additionally, uncertainty can be transmitted vertically: if the PIM and/or the platform model is partial, the generated PSM should also be partial.

**Uncertainty in PIM creation.** While working within a particular layer to produce a PIM, the developer may encounter design decisions for which they have insufficient information. For example, while creating a CIM at the topmost architectural layer to model the business context, developers might be uncertain about choosing a scope, i.e., deciding what domain concepts should or should not be considered in the business model. For concepts that are selected for inclusion to the CIM, further uncertainty can arise about how to best capture their interdependencies, e.g., choosing between is-a and has-a relationships.

Explicating such uncertainty can be done using any Articulation operator. For models at the topmost layer (CIMs), the Expand operator will only be used after entering the Articulation stage via the transition newUncertainty. For models at lower layers, the Expand operator can be used if the developer is expanding upon partial models generated from vertical transformations. During the Deferral stage, the PIM can be used as a basis for the derivation of alternative representations (as partial models) using Transform and can be the subject of verification for structural correctness using the operator Verify. The results of verification can be used to do further diagnosis in the Resolution stage, while additional information can be employed to apply the operators Decide and Constrain.

**Uncertainty in derivative representations.** MDA allows the use of horizontal transformations within an architectural layer, in order to facilitate the derivation of alternative representations of existing PIMs. A different representation might expose modelling options and alternatives that were not easily discernible in the original format. This is possible at all three architectural layers.

Uncertainty can be articulated using the MakePartial operator. The Expand operator can also be used since a derived representation can result from applying the Transform operator to a partial PIM. Further insights can be gained by invoking the Verify operator and the diagnostic operators.
The operator **Transform** can be used to generate additional derivative models. Since alternative representations are derivative artifacts, developers may prefer to avoid invoking resolution operators such as **Decide** and **Constrain** for them. Instead, uncertainty can be propagated back to the PIM using the **Transform** operator. Decisions should then be made on the partial PIM and new versions of the derived representations generated afresh.

**Uncertainty about the content of the platform model.** Vertical PIM to PSM transformations require models that describe the target platform as input. Developers could face uncertainty about the contents of such platform models if developers lack familiarity with a newly adopted platform, if the construction of the platform takes place in parallel with the construction of the application, or if the platform is configurable but the developers are unsure about choosing configuration options. Since the implementation layer is at the lowest level of abstraction, this issue only affects the top two architectural layers.

Since platform models are not generated using higher level models, the Articulation of uncertainty for them is identical as with CIMs, described above. Since platform models do not partake in horizontal transformations, it is not meaningful to apply to them the **Transform** operator for such transformations. However, uncertainty in the platform obviously affects downstream artifacts. Since the correctness of platform models is crucial for PIM/PSM transformations, developers may rely heavily on the **Verify** operator and the respective diagnostic operators (**GenerateCounterExample**, etc.). Resolution of uncertainty can be accomplished with any of the relative operators but, since correctness is a major concern, developers may end up relying more on the **Constrain** operator.

**Uncertainty about platform selection.** If multiple platform models are available, developers might be unsure which ones to choose. Such uncertainty could be caused by a rapidly changing software ecosystem, due to shifting managerial decisions about prioritizing technological platforms, or due to the complexity arising from working with different subcontractors. This issue affects the top two architectural layers.

Uncertainty about choosing a platform is articulated using the **Construct** operator to create a partial model encoding the available options. To handle the case where the set of possible platforms needs to be expanded or contracted, developers can invoke the **Deconstruct** operator, perform any additions or removal and then invoke **Construct** again. During the Deferral stage, developers can use the **Verify** operator to perform sanity checks on the partial platform model. The diagnostic operators and the operator **Constrain** can then be used to appropriately limit the set during the Resolution stage.

**Uncertainty in transformation models.** Uncertainty can also arise during the construction of various model transformations used in MDA. For example, given the task of translating a higher-level concept in a lower level platform, a developer might be uncertain about selecting a target lower-level concept.

Since MDA transformations are themselves models, expressing and resolving uncertainty is accomplished using the Articulation and Resolution operators, respectively. However, there is currently no technique for executing partial transformation models. Strüber et al. have studied transformations that contain variability [Strüber et al., 2015]. Since, as discussed in Section 3.5.3, uncertainty and variability share some conceptual characteristics, their work may provide a way to leverage uncertainty in transformation models during the Deferral stage.
Uncertainty and vertical transformations. The PIM/PSM transition between MDA architectural layers causes uncertainty to be propagated across layers. We distinguish two propagation directions: downstream and upstream. Uncertainty can be propagated downstream, i.e., from higher to lower layers, during the application of automatic transformations. Uncertainty can also be propagated upstream, i.e., from lower to higher layers, as a result of insights gained at lower layers that some upper-layer decisions that were considered certain need to be re-examined.

Downstream propagation of uncertainty occurs when either the PIM or the platform model are partial, and thus the resulting PSM should also be partial. The propagation occurs during the Deferral stage of the management of the input artifacts, using the Transform operator to apply vertical transformations.

Upstream propagation of uncertainty occurs when insights are gained at lower levels of abstraction that have significant impact on higher-level software artifacts. For such propagation to be possible vertical transformations must be bidirectional [Stevens, 2008]. Such propagation is highly non-trivial and is the cause of additional uncertainty. For example, there may be several ways to change a logical model to improve the architectural design based on insights gained at the implementation level. Eramo et al. have studied the introduction of uncertainty in models from bidirectional transformations and have built a prototype implementation [Eramo et al., 2014].

In addition to the propagation of uncertainty, uncertainty resolution can also be propagated either upstream or downstream. Salay et al. have developed an approach for propagating uncertainty resolutions across models, provided that appropriate traceability links are created and maintained [Salay et al., 2013a].

Other sources of uncertainty. Additional sources of uncertainty can include uncertainty in language (i.e., metamodel) definition, uncertainty about tool selection, etc. We focus on uncertainty in the MDA artifacts themselves (i.e., CIMs, PIMs, PSMs, platform models) since other sources of uncertainty are ultimately also manifested in them. Therefore, even though the above list is not complete, it contains sources of uncertainty that uncertainty management in the context of MDA must necessarily address.

Lessons learned. We have applied the uncertainty management approach to Model Driven Architecture, an engineering methodology for creating software systems. The application of uncertainty management to MDA necessitated the critical examination of the different stages of MDA development in order to identify sources of uncertainty.

Having identified the main MDA artifacts in which uncertainty arises, we were able to incorporate the Uncertainty Waterfall in their MDA life-cycle. We also identified that depending on the artifact and its MDA context, different uncertainty management operators may be relevant. For example, while uncertainty in PIMs can be articulated with any Articulation operator, the Construct operator is more useful when dealing with uncertainty about the choice of platform. We also found that further research is required for expressing and leveraging uncertainty in partial transformation models. Finally, we identified that effective vertical propagation of uncertainty and uncertainty resolution highlights the need of bidirectional transformations and traceability links.
Chapter 7. A Methodology for Uncertainty Management

7.4 Tool Support

In this section, we present a tool, called MU-MMINT\(^1\), that implements comprehensive management of models with uncertainty.

MU-MMINT is an Eclipse-based tool for model management [Bernstein, 2003] of partial models. It was created as an Integrated Development Environment (IDE) that bundles various uncertainty management operators in one coherent unit. We illustrate the main features of MU-MMINT using the PtPP motivating example, introduced in Chapter 1.

The workspace of MU-MMINT is an interactive megamodel [Bézivin et al., 2004], shown in the left panel of Figure 7.5, which allows modellers to create, manipulate and interact with partial models using the uncertainty management operations from the Uncertainty Waterfall, shown in Figure 7.4.

**Articulation stage.** Articulating uncertainty in MU-MMINT is done using the MakePartial and Expand uncertainty operators. We illustrate this in PtPP. Initially, the team’s design is separated into known and unknown parts, as shown in Figure 1.2(a). In MU-MMINT, developers can explicate this information in a single partial model \(M_{p2p}\), shown in the middle and right panels of Figure 7.5.

In MU-MMINT, the May formula is modelled graphically using the uncertainty tree, shown in the right panel of Figure 7.5. The uncertainty tree consists of a list of decision elements, each of which can have any number of children representing mutually exclusive alternative solutions. For \(M_{p2p}\), the tree contains the three decisions listed in Figure 1.2(a). For each decision, the team explicates the possible solutions. For example, the decision about the policy when a download completes involves selecting among three alternative solutions, described in Chapter 1: “benevolent”, “selfish”, and “compromise”.

Assuming that the uncertainty tree of a given partial model \(P\) has \(k\) decisions \(\{D_1, ..., D_k\}\), that a given decision \(P_x\) has \(n\) alternative solutions \(\{A_{D_1}^x, ..., A_{D_n}^x\}\) and that a given alternative solution \(A_{D_y}^x\) has \(l\) model elements, the May formula \(\phi_P\) of \(P\) is: \(\phi_P = \bigwedge_{x=1}^{k} \phi_{D_x} \), where \(\phi_{D_x} = \text{Choose}(\phi_{A_{D_1}^x}, ..., \phi_{A_{D_n}^x})\), where Choose is a boolean function that returns True if exactly one of its arguments is True. In turn \(\phi_{A_{D_y}^x} = \bigwedge_{z=1}^{l} u_z\), where \(u_z = e_z\) if \(e_z \in A_{D_y}^x\) and \(u_z = \neg e_z\) if \(e_z \in A_{D_y}^x - A_{D_w}^x\) for \(w \neq y\) and \(e \in P\). An example of this construction is given in Appendix D.2.

\(^1\)Available at: \url{http://github.com/adisandro/mmint}
The middle panel shows the graphical part of the $M_{p2p}$ partial model. It consists of a diagram expressed in the language of partialized state machines, shown in Figure 3.3, that includes Maybe-annotated elements. These elements reify the various alternative solutions and are included in the final version of the model only if their respective solution is selected. For example, the state Finishing and its associated transitions are annotated with (M) and [Compromise] to indicate that they are part of that particular solution to the policy decision.

To further support the articulation process, MU-MMINT supports highlighting the elements reifying a particular solution, as shown in Figure 7.6. MU-MMINT can also highlight the alternative resolutions of a decision point, using different colours, as shown in Figure 7.7. These features allow developers to quickly examine the various possibilities in the partial model.

**Deferral stage.** MU-MMINT implements the Verify operator that allows users to check syntactic properties, such as the property $P_2$ (“no two transitions have the same source and target”) in the PtPP example. If the result of property checking in Maybe or False, MU-MMINT allows users to invoke the GenerateCounterExample operator. For example, Figure 7.8 shows how MU-MMINT generates the concretization in Figure 1.4(c), as a counterexample for $P_2$. MU-MMINT contextualizes the counterexample with respect to the original partial model by greying out unused Maybe elements.

MU-MMINT also implements the Transform operator, using the Henshin graph transformation language and engine [Arendt et al., 2010] to implement lifting as described in Chapter 5. However, support for the Transform operator in MU-MMINT is limited, due to the limited expressiveness of the uncertainty tree. Specifically, lifted transformations do not necessarily produce partial models whose May formulas are expressible as uncertainty trees. Therefore in MU-MMINT, the May formula of partial models created as output by lifted transformations is not shown to users graphically. Instead, it is stored as a raw SMT-LIB [Barrett et al., 2010] string in the workspace megamodel.
Figure 7.7: Highlighting alternatives for the decision “What happens when a download is completed?” in MU-MMINT. *Maybe* elements realizing the “benevolent” scenario are coloured in blue, those realizing the “selfish” scenario – in green, and those realizing the “compromise” scenario – in red.

**Resolution stage.** In addition to the diagnostic operator *GenerateCounterExample* described in the previous paragraph, MU-MMINT implements the *Decide* and *Constrain* operators.

Modellers can invoke the *Constrain* operator to restrict the possible concretizations of the partial model by enforcing a property of interest thus eliminating some design alternatives. MU-MMINT then puts the partial model in Propositional Reduced Form (PRF), automatically recalculating the uncertainty tree and updating the diagrammatic part of the partial model.

Once the modeller has enough information to make a decision, she can invoke the *Decide* operator by selecting the desired alternative solution in the uncertainty tree. This triggers batch applications of *Keep* and *Drop* atomic operations, described in Section 6.1.1. Specifically, *Maybe* elements that reify her chosen solution are kept, and turned into regular model elements, while *Maybe* elements reifying alternative solutions are removed from the model. Additionally, MU-MMINT removes the resolved decision from the uncertainty tree.

MU-MMINT also maintains full traceability between different versions of the partial model in its workspace, as shown in the left panel in Figure 7.5. This allows modellers to easily undo uncertainty resolutions in case they want to revisit certain design decisions (cf. the transition *undoResolution* in the Uncertainty Waterfall).

MU-MMINT was developed in Java by extending MMINT, an interactive environment for model management [Bernstein, 2003] developed at the University of Toronto [Di Sandro et al., 2015]. MMINT consists of 140 KLOC, 80% of which is automatically generated. MU-MMINT is implemented by an additional 10 KLOC, 60% of which is generated. MMINT uses the Eclipse modelling Framework (EMF) [Steinberg et al., 2008] to express models and the Eclipse Graphical modelling Framework (GMF) [Gronback, 2009] for creating graphical editors. MU-MMINT extends MMINT’s data model with EMF structures for uncertainty-related constructs. It also hooks specialized GMF diagram el-
Figure 7.8: Visualizing a counterexample to property $P_2$ in Mu-MMINT.

The main limitation of Mu-MMINT is the expressiveness of the visual syntax used to represent uncertainty in partial models. Our original intent was to realize the graphical syntax introduced in [Famelis, M. and Santosa, S., 2013] which was designed using the theory of visual notations developed by D. Moody [Moody, 2009]. This was not possible because of our reliance on GMF. While GMF allows easy integration of existing model editors to Mu-MMINT, it only supports a limited visual vocabulary. To overcome this, we introduced the concept of the uncertainty tree, as shown in the right panel of Figure 7.5. This approach attempts to ameliorate the limitations of GMF by using a separate dialog, exclusively dedicated to modelling uncertainty at a higher level of abstraction. This idea followed from a preliminary empirical study with human participants [Famelis, M. and Santosa, S., 2013] that pointed us to the need to elevate decisions and alternative solutions to first class concepts in partial modelling. The uncertainty tree approach is less expressive than the fully fledged propositional logic, which results in problems in supporting the Transform operator. However, since it has been shown that humans can graphically create and manipulate many useful formulas [Czarnecki and Wasowski, 2007], this represents a trade-off between usability and expressive power. Resolving this trade-off is ultimately dependent on the usage-specific requirements of the context in which Mu-MMINT is deployed.

7.5 Related Work

The management of uncertainty in the software lifecycle ultimately involves the systematic management of a representation of a set of possibilities. A similar need to manage the lifecycle of abstractions of sets of artifacts arises in related software engineering disciplines such as variability management and design space exploration.
Design space exploration (DSE) [Saxena and Karsai, 2010; Apvrille et al., 2006] is a technique for systematically traversing a space of design choices in order to identify functionally equivalent designs that best satisfy a given set of desired constraints. The DSE process follows a series of steps that is typically described as follows: (1) A reference model is created to represent the requirements and constraints of the desired design. (2) Using the reference model, the developer derives a set of criteria with which to evaluate designs. (3) A model of the design space is created. (4) The developer employs various algorithms to traverse the design space and generate candidate designs. Tools such as Alloy can be used to automate this step [Kang et al., 2011]. (5) Each candidate design is measured against the evaluation criteria. (6) Desirable candidate designs are chosen based on whether they optimally satisfy the evaluation criteria.

The DSE process has certain similarities with the Uncertainty Waterfall: the first two steps resemble the mental process by which a developer identifies a source of uncertainty; the third step corresponds to the Articulation stage of uncertainty management and the final step – to the Resolution stage. The intermediate stage between Articulation and Resolution differs between DSE and uncertainty management. In DSE, decisions are not deferred; rather they are algorithmically explored to produce candidate designs. However, in uncertainty management, the goal is to allow developers to avoid making such decisions. Thus, the focus during this stage is in creating support for lifted engineering tasks that allow developers to continue working with their artifacts without resolving uncertainty.

The goal of variability management [Pohl et al., 2005] is to create and maintain families of products, called Software Product Lines (SPLs), that share a common set of features. In direct analogy to the Uncertainty Waterfall, we can identify three stages of SPL lifecycle: Creation, Maintenance, and Configuration. During SPL creation, the aim is to develop a set of reusable assets that can be combined to produce individual products [Gomaa, 2004]. A common task during this stage is to reverse engineer a SPL from a set of existing products that share functionality, albeit in an ad-hoc way. To address this, various techniques have been developed such as feature location [Rubin and Chechik, 2013], clone detection [Roy et al., 2009] and others. During the Maintenance phase, variability-aware techniques are applied to SPLs in order to manipulate the entire set of products without having to enumerate it. Such techniques include model checking [Classen et al., 2010], type checking [Kästner et al., 2012], testing [Kästner et al., 2012], model transformations [Salay et al., 2014; Famelis et al., 2015b], and others [Midtgaaard et al., 2014; Thüm et al., 2012]. During the Configuration stage, individual products are derived from the SPL to address individual customer needs. This is typically done by configuring a feature model that expresses the allowable combinations of features [Schobbens et al., 2006]. This can be done either at once or in stages [Czarnecki et al., 2004].

The main methodological difference between variability management and uncertainty management is that SPLs represent a long term commitment to supporting and maintaining a product family. It is thus not meaningful to talk about a “Variability Waterfall”: Configuration is not the ultimate endpoint of variability management. Rather, it represents a set of tasks that developers expect to perform often during the lifetime of a SPL. In contrast, partial models are transient artifacts. The Resolution stage of the Uncertainty Waterfall is an endpoint that represents the expectation that uncertainty is ultimately removed from software artifacts. Put simply, partial models are built to throw away, whereas SPLs are built to last.
Chapter 7. A Methodology for Uncertainty Management

7.6 Summary

In this chapter, we have investigated the implications of uncertainty management on software engineering methodology. We have introduced the Uncertainty Waterfall model as a high level abstraction of the lifecycle of uncertainty in software artifacts. The Uncertainty Waterfall model consists of three stages: 1. during the Articulation stage, developers construct partial models to express their uncertainty, 2. during the Deferral stage, the degree of uncertainty in software artifacts remains unchanged, while developers use lifted versions of existing operations to perform engineering tasks, 3. finally, during the Resolution stage, new information is incorporated to resolve uncertainty by refining partial models. Developers transition between these stages based on their engineering needs and the absence or presence of sufficient information to perform these needs.

We have used the Uncertainty Waterfall to contextualize the various techniques presented throughout the thesis. We have summarized each technique as an “uncertainty management operator”, and defined its place within the uncertainty lifecycle in terms of its inputs, outputs, usage context, and conditions prior to and after their invocation. The techniques for modelling uncertainty presented in Chapter 3 were cast as operators used during the Articulation stage. Techniques for performing verification of partial models, presented in Chapter 4, and for transforming partial models, presented in Chapter 5, were used to define the operators of the Deferral stage. Finally, techniques for refining partial models, presented in Chapter 6, as well as techniques for performing diagnosis, presented in Chapter 4, were contextualized as operators used during the Resolution stage.

To illustrate our approach, we have investigated uncertainty management in the context of the Model Driven Architecture (MDA), a software methodology that uses models as the primary development artifact. To do this, we critically examined the different stages of MDA and identified sources of uncertainty. We then illustrated how the different uncertainty management operators can be used to manage this uncertainty within the MDA lifecycle.

Finally, we described Mu-MMINT, an interactive Eclipse-based tool that implements uncertainty management. In Mu-MMINT, implementations of uncertainty management operators from the Uncertainty Waterfall are bundled in a coherent Integrated Development Environment.

In Chapter 8, we illustrate uncertainty management as described by the Uncertainty Waterfall model in fully worked-out examples.
Chapter 8

Worked-Out Examples

In this chapter, we present non-trivial worked examples to illustrate uncertainty management in complex software development settings. We structure the presentation of each example along the workflow defined by the Uncertainty Waterfall model defined in Chapter 7.

In Section 8.1, we present an example of managing uncertainty caused by multiple alternative fixes to a bug in UMLet, an open source software project. In Section 8.2, we illustrate uncertainty management in an example that uses the Object-Relational Mapping transformation benchmark, presented in Section 5.4. We summarize our findings in Section 8.3.

8.1 UMLet Bug #10

On March 8th, 2011, a software developer under the alias “AFDiaX” submitted a bug report to the issue tracker of UMLet, an open source Java-based UML drawing tool [Auer et al., 2003]. The bug report, originally posted on Google Code and since migrated to GitHub as Bug #10, stated:

- copied items should have a higher z-order priority

What steps will reproduce the problem?
1. Copy an item (per double-click)
2. Click on the area where the original and the copy are overlapping
3. Move the mouse

What is the expected output? What do you see instead?
Expected: The new copy should be moved
Instead: The original item is moved

Type-Enhancement Component-UI OpSys-All Priority-Low Usability

That is, if the user copies and then pastes an item within the editor at a location where it overlaps with other existing items, the system does not recognize it as the topmost item, i.e., it does not give it “z-order priority”.

In this section, we use this real world bug to illustrate explicit uncertainty management. Specifically, we create a fictional but realistic scenario in which the maintainer of UMLet attempts to create a fix for bug #10. In our scenario, the maintainer is a practitioner of model-driven software development that uses models as the main artifact for development, relying on code generation to derive the Java implementation.

In order to fix the bug, the maintainer modifies the UML Sequence Diagram modelling the behaviour of the system. However, soon after she realizes that her fix created additional problems because she modified the sequence diagram without properly synchronizing it with the structural aspects (e.g., classes) of the system. This causes her model to violate certain constraints required by the code generator. In order to resolve these constraint violations, she uses an automated technique that generates alternative model repairs [Mens and Van Der Straeten, 2007].

In our scenario, uncertainty arises when the maintainer is unsure about which of these subsequent repairs to choose because their relative merits are unclear. She would thus like to reason with the set of alternative repairs to help her make the choice and possibly even defer the decision until more information is available. In the rest of this section, we show how uncertainty management can be deployed to help the maintainer, illustrating the use of partial modelling techniques throughout the different stages of the Uncertainty Waterfall.

**Description of bug #10 and the maintainer’s bug fix.** In the version of UMLet that was current at the time that the bug was reported\(^2\), the paste functionality was implemented by instantiating the class `Paste` and invoking its `execute` operation. Figure 8.1 shows a fragment of the sequence diagram, generated from the code using the Borland TogetherJ tool\(^3\). The fragment shows `execute` with the circled portion representing the fictional bug fix that the maintainer creates.

Although UMLet has 214 classes in total, we restrict ourselves to a slice that is relevant to the `Paste` class consisting of 6 classes (in addition to Java library classes). These have a total of 44 operations, out of which 13 are invoked in `Paste`. The relevant slice of the UMLet model is captured by model K0, shown in the Appendix Figure D.1. K0 consists of 63 EMF [Steinberg et al., 2009] model elements, out of which 43 are EMF references.

In Figure 8.1, the `for` loop statement block iterates through every item in the clipboard, indexed by variable \(e\). First, each item’s \((x, y)\) coordinates in the editor window are identified (messages 1.36-1.38). The item is then added as an element to the editor window, represented by the object `pnl`, at the coordinates \((x, y)\) of the drawing plane (messages 1.40-1.41).

The bug is caused because when an item is added to a `DrawPanel`, its order in the stack of other items at position \((x, y)\), i.e., its “z-order”, is not set to 0 by default. In our scenario, the maintainer fixes the bug by creating a transient object `positioner` (message 1.42). The `positioner` has a method `moveToTop(e)` that is invoked to place the item on top of others, using the library operation `setComponentZOrder` from the Swing graphical framework (messages 1.43-1.44). In the diagram, the bug fix is shown encircled by a dashed line.

\(^2\)Revision 59 on Google Code, since then migrated to GitHub and available at: [https://github.com/umlet/umlet/commit/f708f57a1fa9f988b3b603e583761e9887ea717ef3](https://github.com/umlet/umlet/commit/f708f57a1fa9f988b3b603e583761e9887ea717ef3), URL accessed on 2015-10-22.

8.1.1 Articulation stage

The fix to bug #10 created by the maintainer is conceptually correct but it violates two consistency rules, defined in [Van Der Straeten et al., 2003], that are required for code generation:

- **ClasslessInstance**: Every object must have a class.
- **DanglingOperation**: The operation used by a message in a sequence diagram must be an operation of the class of the receiving object.

In the maintainer’s bug fix, shown in Figure 8.1, the positioner object violates **ClasslessInstance** because it is not associated with any class. Additionally, the message 1.43 in which the operation **moveToTop** is invoked violates **DanglingOperation** because it is not in **positioner**’s class (since **positioner** has no class).

In order to resolve these consistency violations, the maintainer uses an automated technique that generates alternative model repairs [Mens and Van Der Straeten, 2007]. The technique proposes the following repair strategies for **ClasslessInstance**:

- **RC1**: Remove the object.
- **RC2(obj)**: Replace the object with an existing object **obj** that has a class.
Table 8.1: UMLet example details (cf. Tables 4.2 and 4.3).

<table>
<thead>
<tr>
<th>Value Corresponding experimental category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model size (#Elements)</td>
</tr>
<tr>
<td>115 M</td>
</tr>
<tr>
<td>Set of Concretizations</td>
</tr>
<tr>
<td>44 M</td>
</tr>
</tbody>
</table>

- **RC3(cls)**: Assign the object to the existing class cls.
- **RC4**: Assign the object to a new class.

For **DanglingOperation**, it proposes the following repair strategies:

- **RD1**: Put the operation into the receiving object’s class.
- **RD2(op)**: Change the operation to the operation op that is already in the receiving object’s class.
- **RD3**: Remove the message.

Since the strategy RC1 deletes the object it can only be combined with the strategy RD3, that also deletes the message.

Applying these repair strategies to UMLet results in a set of alternative repairs. Specifically:

- The object **positioner** can be removed (RC1), can be replaced by one of the 5 existing objects (RC2), can be designated as a separate instance of one of the existing 5 classes (RC3), or can be an instance of an altogether new class (RC4).

- The operation **moveToTop** can be removed (RD3), or if **positioner** is assigned a class, it can be either added it (RD1) or it can be swapped for one of the existing 10 operations, depending on which class **positioner** was assigned to (RD2).

In total, there are 44 possible valid repair combinations, listed in the Appendix Table D.1.

Using this list, the maintainer is able to express her uncertainty using the Articulation operator **MakePartial**. Specifically, she uses Mu-MMINT to edit the UMLet model K0 and create a partial model K1. The diagram of K1 is shown in the Appendix Figure D.2 and consists of 115 EMF model elements, 52 of which are annotated with **Maybe**. In K1 all the model elements of the maintainer’s fix (shown encircled by a dashed line in Figure 8.1) become **Maybe** since they are present in some alternatives and absent in others. The May formula of K1 is modelled as an uncertainty tree two decision points, one for each consistency constraint, that have four and three alternative solutions respectively, as described earlier. As a result, K1 has 44 concretizations, each corresponding to a valid repair combination.

We summarize the details about the size of the generated partial model in Table 8.1, comparing the size of the UMLet slice with the experimental categories in Table 4.2 and Table 4.3.

### 8.1.2 Deferral stage

Having constructed the partial model K1, the maintainer can defer the selection of one of the possible bug fixes until she has sufficient information about them. In the meantime, she is able to perform other tasks, without having to artificially remove uncertainty. In our scenario, the maintainer wishes to reassure the users of UMLet that the changes introduced by her bug fix do not affect existing functionality. In
particular, she focuses on the behaviour of the *paste* command, wishing to show that the property $U_1$ holds, regardless of the details about how bug #10 was fixed:

**$U_1$:** *Whenever an item is pasted, a new item is created in the editor window.*

For her model to satisfy property $U_1$, it must have: (a) the message *paste* from the *Paste* class to the *Clipboard* class to obtain the pasted *GridElement* $e$, (b) the message *cloneFromMe* to $e$ in order to create a new *GridElement* copy to be pasted, and finally (c) the instantiation of a new *AddElement* command object to add the item to the editor window. The maintainer uses the operator *Verify* to show that this is the case. To do this, she first encodes the property $U_1$ in logic, creating the formula $\phi_{U_1}$. The grounded version of $\phi_{U_1}$, expressed in SMT-LIB [Barrett et al., 2010] is shown in the Appendix Figure D.5. She then invokes the *Verify* operator with $K_1$ and $\phi_{U_1}$ as inputs.

In **MU-MMINT**, the invocation of *Verify* follows the decision procedure described in Chapter 4. First, the queries $F^+(SD, K_1, U_1)$ and $F^-(SD, K_1, U_1)$ are constructed, where “SD” stands for the well-formedness constraints of sequence diagrams. Their encoding for this example is shown in the Appendix Figure D.4. The two queries are then given as inputs to the Z3 SMT solver [De Moura and Bjørner, 2011] and the results are interpreted according to Table 4.1. In our scenario, the *Verify* operator returns *True*, indicating that all concretizations of $K_1$ satisfy the property. This is reasonable, since neither the bug fix, nor the automatically generated consistency repairs repairs affected that part of the model.

To triangulate the experimental findings described in Section 4.3.5, we compared this method of performing *Verify* with the “classical” approach of checking the property for each concretization separately, as described in Section 4.3.1. The comparison showed that the version of *Verify* using partial models was 3.84 times faster than the alternative.

Since checking property $U_1$ yielded *True*, the maintainer is able to reassure her users that the alternative bug fixes do not break the paste functionality. Having determined that $U_1$ is not a factor in deciding how to resolve uncertainty, she now returns her attention to the property $U_2$:

**$U_2$:** *Each item that is pasted from the clipboard must have z-order=0.*

It was the violation of this property that originally caused bug #10.

In her model, ensuring that pasted elements are assigned the correct z-order is done by invoking the method *setComponentZOrder* of the class *DrawPanel* with the item $e$ and the z-order 0 as parameters. Again, the maintainer invokes the operator *Verify* to check that this is the case. She encodes $U_2$ in logic, creating the formula $\phi_{U_2}$, the grounded version of which is shown in the Appendix Figure D.3 along with the relevant well-formedness condition. In this case the operator *Verify* yields the result *Maybe*, indicating that some but not all of the concretizations satisfy the property. Compared to classical verification, checking $U_2$ with partial models was 1.95 times faster.

### 8.1.3 Resolution stage

The maintainer is alarmed by the verification result since it indicates that not all concretizations of $K_1$ actually fix bug #10. To understand why that is the case, she uses the operator *GenerateCounterExample* with $K_1$ and $U_2$ as inputs. The operator finds the concretization $K_2$, shown in the Appendix Figure D.6, which is the result of applying the consistency repairs RC1 and RD3, i.e., deleting the model elements the maintainer added to $K_0$ to fix the bug in the first place.
To ensure that bug #10 is fixed, the maintainer must refine her partial model, i.e., reduce its set of concretizations to the subset that satisfies U2. To accomplish this she invokes the operator **Constrain** with K1 and U2 as inputs. As described in Section 6.1.2, this entails combining U2 with the May formula of K1 and then putting the result in Graphical Reduced Form (GRF). In the resulting partial model K3, the diagram of which is shown in the Appendix Figure D.7, the *positioner*, as well as the messages *moveToTop*, *setComponentZOrder*, and *new* (which instantiates the *positioner*) are no longer annotated with *Maybe*. This means that they must exist in each concretization of K3. However, all the edges that have one of these elements as their source are annotated with *Maybe*. For example, the *positioner* object has 6 outgoing class reference edges, all of which are annotated with *Maybe*. This is because the decision about what class this object is an instance of has yet to be made. The allowable combinations of *Maybe* elements are captured in the May formula $\phi_{K3}$. Specifically, $\phi_{K3}$ encodes the repair combinations 2 through 44 from the Appendix Table D.1.

To triangulate the experimental findings described in Section 6.2.3, we compared alternative ways of computing the GRF of K3, as described in Section 6.2.1. We found that computing the GRF using Algorithm 3 than using Algorithm 1 results in a 0.45 slowdown. This is comparable with the finding of a 0.525 $S_4/3$ slowdown (cf. Figure 6.1(a)) for a model with size in the M category and size of set of concretizations in the M category.

At this point, the maintainer has created a partial model that encodes a set of possible models, all of which are both valid fixes to bug #10 and consistent with the constraints imposed by the code generator. She can choose to again defer making a decision, thus entering a second Deferral stage, or to further resolve uncertainty if she feels she has adequate information to do so.

In our scenario, she chooses the combination of repair options RC4, RD2, which assigns *positioner* to a new class *NewClass* that also has the method *moveToTop*. To accomplish this, she invokes the operator **Decide**, thus creating the concrete model K4, shown in the Appendix Figure D.8.

### 8.1.4 Summary and lessons learned

**Summary.** We summarize the UMLet example in Figure 8.2, which superimposes the models and invocations of uncertainty management operators in the scenario over the Uncertainty Waterfall model.

At the start of the scenario, the maintainer of UMLet received a bug report saying that pasted elements in the UMLet editor do not have the correct z-order. To fix the bug, she identified a slice of UMLet that contained the bug, represented by the model K0. In fixing the bug, she realized that her changes caused two consistency constraints required by the code generator to be violated. She used an automated technique for generating repairs to these constraints, which resulted in a set of 44 alternative ways to fix the bug. Not having enough information to choose between them, she entered the uncertainty Articulation stage and created the partial model K1 using the operator **MakePartial** on K0. Having expressed her uncertainty, she entered the Deferral stage, where she used the **Verify** operator to check properties of her partial model.

In the process, she found out that K1 contained concretizations which did not really fix the original bug. She proceeded to remove them from her model, thus entering the Resolution stage. She used the **GenerateCounterExample** operator to diagnose why some of the concretizations were not valid fixes. The counterexample K2 showed her that a particular combination of strategies to repair the consistency violations effectively undid her original repair. To address this, she used the operator **Constrain** to remove the offending concretizations, thus creating the partial model K3 that only contained valid fixes.
Figure 8.2: Overview of the UMLet example, superimposed over the Uncertainty Waterfall model.

Finally, she chose a particular combination of consistency repairs and used the Decide operator to resolve all uncertainty in her models, creating the model K4.

**Lessons learned.** The UMLet example allowed us to work a realistic, non-trivial example through the stages of the Uncertainty Waterfall. Two main lessons emerged:

1. Articulation of uncertainty requires automated support. During the Articulation stage, we had to express a large combination of possible repairs as a partial model. Even with the tooling support provided by Mu-MMINT, the manual construction of the partial model K1 using the MakePartial operator was tedious and error-prone. In order for the effort expended in constructing a partial model to be outweighed by the benefits, we need to create additional automated support. This could be achieved using techniques such as Design Space Exploration [Saxena and Karsai, 2010], by integrating partial modelling into sketching tools [Mangano et al., 2010; Wiest et al., 2015], or by mining the social context of development, such as online discussions between developers [Mashiyat et al., 2014]. We discuss these potential research directions in more detail in Section 9.2.

2. Rigid separation of verification from diagnostic operators is counter-intuitive. In the Uncertainty Waterfall model, the Verify operator is placed in the Deferral stage, whereas diagnostic operators, such as GenerateCounterExample, are placed in the Resolution stage. There are good reasons for this: during Resolution uncertainty is reduced, whereas during Deferral its level remains constant. However, since in practice verification and diagnosis are closely intertwined, it is important to emphasize that the transition between of stages is not as rigid, even when taking into account the backward transitions in Figure 7.3.

In addition to the above, the UMLet example allowed us to triangulate the experimental findings from Chapters 4 and 6. We summarize our observations in Table 8.2. In Table 8.2, each speedup in the UMLet example is shown next to the corresponding experimental result for models of size $M$ with $M$ size of concretizations. The observation for U1 is compared with the average of property checks
Table 8.2: UMLet example observed speedups, compared with experimental observations.

<table>
<thead>
<tr>
<th></th>
<th>Verification Speedup (E2/E1)</th>
<th>GRF Construction Speedup (E4/E3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMLet example</td>
<td>Experiments</td>
<td>UMLet example</td>
</tr>
<tr>
<td>Property U1</td>
<td>3.84</td>
<td>2.85</td>
</tr>
<tr>
<td>Property U2</td>
<td>1.95</td>
<td>1.49</td>
</tr>
</tbody>
</table>

that returned True or False, whereas the observations for U2 are compared with those for which the property check returned Maybe. For verification, the speedups that we observed are slightly higher than the results observed experimentally using randomly generated inputs (cf. Figure 4.1). For GRF construction, the observed slowdown was slightly slower (cf. Figure 6.1). We believe that there are two explanations for these findings: 1. Compared to the generated graphs used for our experiments, the structure (and therefore the resulting propositional encoding) of the UMLet models is not random. 2. For our experimental evaluation, we used untyped directed graphs because the absence of metamodel constraints makes the problem harder for the solver to solve. On the other hand, we encoded additional metamodel constraints for the UMLet example. Overall, these observations lead us to conclude that the experimental results detailed in Sections 4.3.5 and 6.2.3 constitute a lower bound of the performance of our approach.

8.2 Petri Net Metamodel

In this section, we take the view of a toolsmith who is tasked with creating a fictional tool, called CONCMod, for modelling concurrent systems. CONCMod uses Petri nets (PTNs), a powerful formalism used widely in this domain, first introduced by Carl Petri in 1962 [Petri, 1962]. A succinct description of PTNs is found in [Jensen and Kristensen, 2015]:

“A Petri net in its basic form is called a place/transition net, or PTN, and is a directed bi-partite graph with nodes consisting of places (drawn as ellipses) and transitions (drawn as rectangles). The state of a Petri net is called a marking and consists of a distribution of tokens (drawn as black dots) positioned on the places. The execution of a Petri net (also called the “token game”) consists of occurrences of enabled transitions removing tokens from input places and adding tokens to output places, as described by integer arc weights, thereby changing the current state (marking) of the Petri net. An abundance of structural analysis methods (such as invariants and net reductions), as well as dynamic analysis methods (such as state spaces and coverability graphs), exist for Petri nets.”

Figure 8.3: Example Petri net token game. Left: Transition t1 is enabled. Right: the Petri net after firing t1.
An example PTN, consisting of two Places and one Transition is shown in Figure 8.3. In the original marking, shown on the left, the Place \( p_1 \) contains one Token. The Transition \( t_1 \) has a single incoming arc, and since that arc has weight 1, and \( p_1 \) has one Token, \( t_1 \) is enabled. When \( t_1 \) fires, the Petri net gets the marking shown on the right. In this new state, the Token in \( p_1 \) has been consumed and because \( t_1 \) has a single outgoing arc with weight 1, a single token has been produced in Place \( p_2 \). Since \( p_1 \) is empty, \( t_1 \) is no longer enabled.

In our scenario, we assume that during the initial design phase of the ConCMOD project, the toolsmith wants to create a metamodel for representing PTNs such as the one shown in Figure 8.3. In this section, we use the development of this metamodel as an example for explicit uncertainty management.

### 8.2.1 Base PTN metamodel

**Articulation stage.** In order to avoid re-inventing a PTN metamodel from scratch, the toolsmith consults a public metamodel repository called Atlantic Metamodel Zoo (AMZ) [Jouault and Bézivin, 2006]. In AMZ, she discovers eight different PTN metamodels. While all metamodels have some basic PTN concepts in common, such as meta-classes for Places and Transitions, they are different in a few significant ways. By inspecting the metamodels, the toolsmith identifies the following design decisions that cause them to diverge:

- **ArcClasses**: Arcs are represented using separate meta-classes.
- **WeighedArcs**: Arc meta-classes have attributes to represent arc weight.
- **Locations**: There is a way to store the location of graphical elements on the diagram.
- **TokenClass**: Tokens are represented using a separate meta-class.
- **Executions**: If a separate token meta-class exists, there is a mechanism for representing “token games”.

Table 8.3 summarizes what decisions are implemented in each of the eight PTN metamodels in AMZ. The case where a metamodel implements a decision is indicated by “✓” and the case where it does not — by “✗”. Metamodels that implement the same decisions differ in minor ways. Specifically, GWPNV1 differs from GWPNV0 by requiring that a PTN must have at least one Place and Transition, GWPNV3 from GWPNV2 — by introducing a superclass for arcs, and PetriNet differs from GWPNV4 by introducing a class Element. In our scenario, the toolsmith decides that these differences are not significant enough for her purposes, and thus chooses to disregard them.

Since each decision is binary and two decisions are dependent on others (WeighedArcs depends on ArcClasses, Executions depends on TokenClass), the toolsmith can create \((2 + 1) \times 2 \times (2 + 1) = 18\) different combinations. However, she does not have enough information to decide which combination is best for ConCMOD. She thus uses the Construct operator to create the partial metamodel N0, the diagram of which is shown in the Appendix Figure D.9. The partial model N0 has a total of 76 elements (52 node and 24 vertex elements), out of which 60 are annotated with Maybe. N0 has 18 concretizations, defined by its May formula, the construction of which is described in Appendix D.2. Compared to the

---


5 The metamodels with the prefix GWPN were created by Guido Wachsmut and Kelly Garces. The metamodel Extended was created by Hugo Bruneliere and Pierrick Guyard. The metamodel PetriNet was created by David Touzet.
Table 8.3: Design decisions in Petri net metamodels from the Atlantic Metamodel Zoo. Each column represents one metamodel.

<table>
<thead>
<tr>
<th></th>
<th>GWPNV0</th>
<th>GWPNV1</th>
<th>GWPNV2</th>
<th>GWPNV3</th>
<th>GWPNV4</th>
<th>GWPNV5</th>
<th>Extended PetriNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArcClasses</td>
<td>☒</td>
<td>☒</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>WeighedArcs</td>
<td>☒</td>
<td>☒</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Locations</td>
<td>☒</td>
<td>☒</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>TokenClass</td>
<td>☒</td>
<td>☒</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Executions</td>
<td>☒</td>
<td>☒</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

categories in Tables 4.2 and 4.3, N0 falls in the S category for model size and the M category for the size of the set of concretizations. Additionally, N0 has more Maybe elements and fewer concretizations than any of the partial models used as input to the ORM benchmark, described in Table 5.1.

**Deferral Stage: Transformation.** Having explicited her uncertainty in the partial model N0, the toolsmith can defer making a decision about how exactly to design the CONC MOD metamodel. In the meantime, she can use N0 to proceed with other development tasks. In our scenario, the focus of the toolsmith is turned to the serialization of PTN models. In order to efficiently store PTN models, the toolsmith wants to use a relational database. She thus needs to create a database schema that allows her to store instances of PTN models.

Mapping the PTN metamodel to a relational database schema is an instance of the well-known “Object Relational Mapping” (ORM) problem. In model-driven software engineering, a typical solution to the ORM problem involves creating a model transformation that takes a class diagram as input and produces a relational schema model as output. In our scenario, the toolsmith uses an ORM transformation published in [Varró et al., 2006] that consists of five layered rewrite rules that, given a class diagram, create a relational schema and traceability links between them. The rules and the input/output metamodels are shown in Appendix C.

The toolsmith invokes the uncertainty management operator Transform with N0 and the ORM transformation as inputs. The result is the partial relational schema model N1, consisting of 30 tables. Overall, N1 has 192 elements (tables, columns, and key references), out of which 184 are Maybe. The runtime of the transformation was 2.00 seconds and the generated SMT-LIB May formula has the size of 23.26 kB.

**Deferral Stage: Verification.** The toolsmith continues working in the presence of uncertainty. Having created a partial relational database schema, she decides to investigate whether her schema allows her to accurately store the graphical diagrams of PTN models. She formalizes this requirement as a property:

**U3:** The database must allow storing the diagram coordinates of every graphical PTN construct.

For her database schema to satisfy U3, for each table $T_i$ in N1 that stores some graphical PTN construct (i.e., the tables Place, Transition, Token, PlaceToTransitionArc, TransitionToPlaceArc) there must exist a table $TL_i$ that maps tuples from $T_i$ to tuples from the table Location. The toolsmith uses the operator Verify to check whether N1 satisfies U3. To do this, she first encodes the property U3 in logic, creating the formula $\phi_{U3}$. The grounded version of $\phi_{U3}$, expressed in SMT-LIB [Barrett et al., 2010], is shown in the Appendix Figure D.11. She then invokes the Verify operator with N1 and $\phi_{U3}$ as
Chapter 8. Worked-Out Examples

inputs. The result of the property check is Maybe, indicating that U3 may not be satisfied, depending on how uncertainty is resolved.

Resolution stage. In order to discover the reason why checking U3 on N1 results in Maybe, the toolsmith uses the diagnostic operators. Using the operator GenerateCounterExample, she discovers that since the Location table is annotated with Maybe, there exists at least one concretization in which it does not exist. Since her primary development artifact is the partial metamodel N0, she uses this new information to refine N0 using the operator Decide to ensure that the class Location is present in all concretizations. The result is the partial metamodel N2, shown in the Appendix Figure D.12.

Additionally, by invoking the operator GenerateDiagnosticCore with N1 and U3 as inputs, she discovers that there are no tables mapping tuples of the tables Token, PlaceToTransitionArc, TransitionToPlaceArc to the table Location. She decides that this is acceptable, since the location of these graphical elements can be derived from Places and the combination of endpoint Place and Transition, respectively.

The toolsmith does not have any more information to fully resolve the uncertainty in N2. Additionally, she receives new requirements for ConcMod, which cause her more uncertainty, prompting her to enter a new uncertainty Articulation stage.

8.2.2 Extended PTN metamodel

Articulation stage. Due to its simplicity, the Petri net formalism lends itself to customization in order to capture domain-specific concerns. This has lead to the proliferation of Petri net extensions, which augment the base language with special-purpose constructs. Examples of such Petri net extensions include Coloured PTNs [Jensen, 1987], Hierarchical PTNs [Fehling, 1993], Prioritized PTNs [Guan et al., 1998], Timed PTNs [Ramchandani, 1974], Stochastic PTNs [Molloy, 1982], and others. In our scenario, the toolsmith learns that her employer is considering allowing such PTN models to be stored in ConcMod. However, she does not have enough information about which domain-specific constructs should be included in the ConcMod metamodel.

Faced with uncertainty about which Petri net extensions to support, the toolsmith uses the operator Expand on the partial metamodel N2, to create the partial metamodel N3. The uncertainty tree of N3, shown in the Appendix Figure D.14, contains several new (binary) design decision points:

- **ArcKinds**: There are different kinds of arcs, such as inhibitor and reading arcs [Murata, 1989].
- **Priority**: Transitions have priorities [Guan et al., 1998].
- **Timed**: Transitions are timed [Ramchandani, 1974].
- **ColouredTokens**: Tokens have values, called “colours” taken from “Colour Sets”, i.e., types [Jensen, 1987].
- **Stochastic**: Each transitions has a “firing rate” that indicates the probability that it will fire at every marking [Molloy, 1982].
- **Guards**: Arcs have guard expressions [Jensen, 1987].
The decisions **Guards**, **WeighedArcs**, and **ArcKinds** depend on the decision **ArcClasses**, while the decisions **Executions** and **ColouredTokens** depend on the decision **TokenClass**. Therefore, the uncertainty tree allows $(1 + 2 \times 2 \times 2) \times (1 + 2 \times 2) \times 2 \times 2 = 360$ concretizations. The model elements required to implement these design decisions bring the total elements of N3 to 117 elements, out of which 94 are **Maybe**. The diagram of N3 is shown in the Appendix Figure D.13. Compared to the categories in Tables 4.2 and 4.3, N3 falls into the M category for model size and is larger than the XL category for the size of the set of concretizations. Additionally, N3 has more **Maybe** elements and more concretizations than any of the partial models used as input to the ORM benchmark, described in Table 5.1.

**Deferral stage.** Having explicated her uncertainty in the partial model N3, the toolsmith again generates a relational database schema using the ORM transformation. She invokes the uncertainty management operator **Transform** again, this time with N3 and the ORM transformation as inputs. The result is the partial relational schema model N4, which has 45 tables. In total, N4 has 293 elements (tables, columns, and key references), out of which 258 are **Maybe**. The total runtime of the transformation was 114.05 seconds and the generated SMT-LIB May formula is 33.78 kB long.

**Resolution stage.** Using lifted operations, the toolsmith can continue working in the presence of uncertainty for as long as necessary. In our scenario, we assume that at some point the toolsmith is able to resolve all of her uncertainty by making all of the design decisions. Specifically, she uses the **Decide** operator with the partial model N3 as input and makes the choice to not implement any of the domain-specific Petri net extensions, to allow executions to be stored, and to use separate arc and token meta-classes. These decisions result in the concrete metamodel N5, shown in the Appendix Figure D.15.

### 8.2.3 Summary and lessons learned.

**Summary.** We summarize the PTN metamodel example in Figure 8.4, which superimposes the models and invocations of uncertainty management operators in the scenario over the Uncertainty Waterfall model.

At the start of the scenario, the toolsmith in charge of creating **ConcMod** aimed to create a metamodel for representing Petri nets. She consulted an open repository of metamodels, where she located eight existing different Petri net metamodels. By analysing these metamodels, she identified five important design decisions that are the cause of the differences between them. Not having enough information to decide how to make these decisions, she used the operator **Construct** to create the partial metamodel N0.

Having articulated her uncertainty in N0, the toolsmith entered the Deferral stage. She applied the ORM transformation to her partial metamodel using the operation **Transform**, to create a derived partial relational schema model N1. She then applied the operator **Verify** to check whether N1 allows storing the diagrammatic locations of Petri net elements. The result of the check was **Maybe**, and prompted her to perform further diagnosis using the diagnostic operators. She discovered that (a) not all concretizations have a dedicated table for storing locations, and (b) the locations of some of the diagrammatic elements can be derived from others.

The first diagnostic insight led her to enter the Resolution stage, where she used the newly acquired information to resolve some of the uncertainty in N0, to ensure that the **Location** table is present in all
Articulation/1

1: doWork
Construct
N0

Articulation/2

2: newInformation
Expand
N2

Deferral/1

N0
Transform
N1
Verify(U3)
Maybe

Deferral/2

N3
Transform
N4

Resolution/1

N0
Resolve
N2

Resolution/2

N3
Resolve
N5

Lessons learned. In the Petri net metamodel example, we worked with a medium-sized model with a large set of concretizations and a large percentage of Maybe elements through the different stages of uncertainty management. Three main lessons arose from this experience:

1. Even though we described the initial articulation of uncertainty in terms of invoking the operator Construct for the set of all possible metamodel designs, in practice, we had to adopt a slightly different approach. Specifically, we manually constructed the partial model N0, using the eight AMZ metamodels and the five design decisions as guidance. This was a different process than both Construct (which is fully automatic but depends on an enumerated set of alternatives) and MakePartial (which is manual but depends on a pre-existing concrete model). Rather, in practice, we ended up adopting elements of both, whereby we manually enumerated a set of possibilities, using the uncertainty tree as a guide.

2. The number of application sites and the number of Maybe elements in the partial models N0 and N3 was significantly larger than that of the models used for the ORM benchmark (described in
Table 5.1). This made the original implementation of our lifted transformation engine extremely inefficient. We were thus forced to revisit it and implement various optimizations that resulted in improvements in both runtime and the size of the generated May formula. These improvements were so dramatic as to render comparison with the benchmark results meaningless. However, during the implementation of the optimizations, we faced significant challenges in correctly engineering complex manipulations of large formulas while working with the application programming interface (API) of the Z3 solver [De Moura and Bjørner, 2011]. This process was extremely error prone and hard to debug, thus exposing some of the practical difficulties of our lifting strategy, that focuses on manipulating a single large propositional expression. We contrast this with our experience implementing lifting for product lines, where individual elements are annotated with “presence conditions” [Salay et al., 2014; Famelis et al., 2015b]. This makes the implementation of lifting significantly easier, since instead of manipulating a single large formula, it uses smaller logical expressions that are localized to specific matching sites.

3. During the first Resolution stage, the toolsmith invoked the Verify operator, which returned Maybe. However, instead of using the result to constrain the partial model, we found that it was more appropriate for the toolsmith to change her expectations about the property. This points to a characterization of properties with modalities, such as “necessary” and “possible”. Responses to property checks can therefore include either refining the underlying model or changing the modality of the property. In current work, we are investigating such property modalities and appropriate responses to the verification of such properties in the context of product lines with uncertainty [Chechik et al., 2016].

8.3 Summary

In this chapter, we have applied the techniques described in the rest of the thesis to non-trivial, fully worked-out examples. We have used the Uncertainty Waterfall model, introduced in Chapter 7, to illustrate two scenarios of explicit uncertainty management.

The experience gained from these examples has illustrated the capabilities of our approach and exposed some of its limitations. In Chapter 9, we build upon this experience to outline potential directions for future research.
Chapter 9

Conclusion

This chapter concludes the thesis. In Section 9.1, we summarize the key points of the thesis and in Section 9.2, we discuss limitations of our approach and outline directions for future work to address them.

9.1 Summary

In this thesis, we have presented an approach for helping developers avoid premature commitments while designing software systems. Such situations arise when developers face uncertainty about the content of their software artifacts. Since existing software engineering languages, tools, and methodologies do not support working in the presence of uncertainty, developers are forced to either refrain from using them or to make provisional decisions and attempt to keep track of them in case they need to revisited. While the first approach leads to under-utilization of resources, the second risks potentially costly re-engineering efforts in case the otherwise artificial removal of uncertainty proves premature.

Instead, we have proposed an approach for deferring decisions which developers are not ready to make. The approach entails carefully managing the lifecycle of uncertainty in software artifacts, while adapting engineering tasks so that they can be performed in the presence of uncertainty. We captured this process in the Uncertainty Waterfall, described in Chapter 7, which is a model for the management of the lifecycle of uncertainty. This lifecycle consists of three stages: Articulation, Deferral, and Resolution. Uncertainty management operations pertinent to each stage were presented in detail in Chapters 3, 4 and 5, and 6, respectively.

In Chapter 3, we presented an approach for articulating uncertainty using partial models, a formalism for compactly and exactly representing sets of possibilities. In partial models, uncertainty about selecting a possible design is expressed by annotating elements of a base model as “Maybe” while a propositional May formula constrains their possible combinations, called concretizations. Creating such combinations is done through a process of structure-preserving refinement, which also forms the basis for defining the formal semantics of partial models. We introduced a partial model construction algorithm and two equivalent “reduced forms”, suited for different usage settings. We also described and evaluated the user-friendliness of the partial model notation and introduced a partialization process for adapting the metamodel of the base language to support expressing uncertainty.

In Chapter 4, we described automated verification for partial models. We defined the meaning of
partial model property checking and explained that because partial models have concretization-based semantics (i.e., are defined through structure-preserving refinement), we scoped our verification technique to checking syntactic properties. We gave a decision procedure for checking such properties and described three kinds of feedback that can be generated in order to help developers perform diagnosis. Through experimental evaluation of our verification approach, we found that partial model verification results in significant speedup compared to classical verification (i.e., checking the property on each concretization separately). We also observed that the speedup increases with greater degrees of uncertainty (i.e., larger sets of concretizations).

In Chapter 5, we introduced the lifting of graph-based model transformations such that they can be applied to partial models. We formally introduced the concept of lifting: applying a transformation to a partial model should produce a new partial model which should be the same as if we had transformed each concretization of the input partial model separately and then merged them back. A corollary to this definition is that existing transformations do not need to be redefined for partial models. Instead, we lifted the semantics of model transformation application. We did this through a three-step process: given a candidate transformation site in a model, we first determine rule applicability using a SAT solver; then we transform the graphical part of the partial model; finally, we transform the May formula to ensure that the correctness of lifting is respected. We showed that our process preserves lifting correctness as well as confluence and termination, two important properties of graph-based model transformations. We applied our approach to the Object-Relational Mapping, a common benchmark for model transformation techniques, and discovered there is a trade-off between transformation runtime and the growth of the May formula of the output.

In Chapter 6, we described how uncertainty is resolved by systematically integrating newly acquired information to partial models. We introduced two approaches to operationalize partial model refinement: manual decision making and property-driven refinement. In the former, the modeller incorporates newly acquired information in the partial model by making decisions about whether to Keep or Drop individual Maybe elements. In the latter, uncertainty is resolved declaratively, by expressing newly acquired information as a property. A refinement is then constructed by appropriately combining the property with the partial model's May formula. Through experimentation, we evaluated two different approaches for performing this construction, and found that each works best for different combinations of model size and size of the set of concretizations.

We combined these techniques in a coherent whole in Chapter 7, where we investigated the implications of uncertainty management on software engineering methodology. To capture the lifecycle of uncertainty in software artifacts we introduced the Uncertainty Waterfall model and used it to contextualize the various uncertainty management operators. To illustrate our approach, we applied it to the Model Driven Architecture (MDA), a software methodology that uses models as the primary development artifact. We critically examined the different stages of MDA and identified sources of uncertainty. We then illustrated how the different uncertainty management operators can be used to manage this uncertainty within the MDA lifecycle. Finally, we described Mu-MMINT, an interactive Eclipse-based tool that implements uncertainty management by bundling the implementations of uncertainty management operators in an Integrated Development Environment for managing partial models.

In Chapter 8, we illustrated the lifecycle of uncertainty management in fully worked-out examples. The experience gained from working these examples through the lifecycle of uncertainty illustrated the capabilities of our approach and exposed some of its limitations. We discuss these in the next section,
along with directions for research to mitigate them.

9.2 Limitations and Future Work

The concern for handling uncertainty permeates software development. In this thesis, we have presented a particular approach for accomplishing this, based on facilitating the deferral of design decisions. We have thus presented a coherent and minimally complete subset of what should become a comprehensive theory of uncertainty management. Below we outline several directions for future work in what promises to be a fruitful area of research.

**Balancing deferral, provisionality, and uncertainty avoidance.** Developers routinely face design-time uncertainty in their day-to-day engineering activities. The virtual omnipresence of functional software in contemporary society proves they are capable of effectively dealing with it. This is done using either of the following: (a) development approaches that avoid uncertainty for as long as necessary by routing around uncertain parts of the developed system, such as lean software development [Poppendieck and Poppendieck, 2003b], (b) iterative methodologies that embrace provisionality, such as Agile Development [Martin, 2003], or (c) ad-hoc handling of provisional decisions based on experience and intuition.

We intend to further investigate the value proposition of explicit uncertainty management with relation to existing software development practices. To accomplish this, we intend to study the historical data of software systems in order to identify cases of catastrophic mismanagement of uncertainty. Subsequently, we will perform post-hoc analysis in order to determine the effectiveness of different uncertainty management strategies. In addition, we intend to study the integration of explicit uncertainty management into lean software development, such as the Scrumban methodology [Ladas, 2009]. This will allow us to either develop mixed strategies, effectively weaving explicit uncertainty management with existing approaches, or to recognize niche contexts in which it is cost-effective.

**Identification of design decisions and elicitation of solutions.** In Section 1.3, we scoped our approach by making two important assumptions: (a) that developers know what they are uncertain about, and (b) that all decision points are “known unknowns”, i.e., closed questions where developers are uncertain about choosing one among a well understood finite set of acceptable solutions. Thus, the uncertainty management operators that we defined for the Articulation stage of the Uncertainty Waterfall in Section 7.2.1 assume as input either a fully enumerated set of possibilities or an informal description of how uncertainty should be expressed in partial models. However, in order to use uncertainty management in the context of real software development, additional support must be provided for (a) identifying that development has reached a critical design decision such that explicit uncertainty management is cost-effective, (b) assessing the impact of uncertainty across multiple artifacts, (c) helping developers elicit a set of acceptable solutions to an open design decision (an “unknown unknown”), and (d) helping encode them in a partial model.

To address the first two points, we intend to study the identification of design decisions and the assessment of their impact, by focusing on the socio-technical context in which such decisions are made. A first step in this direction is to study historical data from existing software projects in order to develop a theory about when major design decisions occur in the lifecycle of software projects. A potential approach is to attempt to identify patterns among changes that had significant impact downstream,
such as changes that resulted in a lot of bugs or changes that required a lot of development effort to be undone. A different approach is to study the social context in which design decisions are made. We discuss this possibility in more detail in the next paragraph.

To address the latter points, we intend to investigate the use of domain space exploration techniques [Saxena and Karsai, 2010] for synthesizing sets of possibly acceptable solutions. This would entail mining the development context to identify a set of acceptability criteria and then generating designs that are equivalent with respect to these criteria. Additional research must investigate ways for presenting these recommendations to the developers in a user-friendly way.

Leveraging the social context of software development. Software development is an inherently social process. For example, the first step for addressing a bug report or a feature request is to start a conversation between the stakeholders of a software project. This way, points of uncertainty in the solution space can be identified and proposals to resolve them can be explored. However, due to the fluid nature of such interactions, it is often hard to understand the state of the discussion and its impact on the quality of actual software artifacts. In [Mashiyat et al., 2014], we proposed to analyze the utterances of uncertainty in conversations to create business indicators and analytics. These can help managers of software development teams assess the state of a project, identify pain points and recommend interventions. The main challenge lies in automatically generating models of the rhetorical structure of conversations. This entails the use of natural language processing techniques to: (a) identify utterances of uncertainty and utterances of proposed resolutions, (b) match each point of uncertainty with its corresponding proposals, (c) recognize when a proposal is accepted as a sufficient resolution of a point of uncertainty. Additionally, we can map utterances of uncertainty to software artifacts and use propagation techniques [Salay et al., 2013a] to help developers articulate and/or resolve uncertainty.

Providing a formal back-end for informal modelling. The empirical work of Marian Petre has shown that one of the most common uses of UML in practice is as an informal “thought tool” during initial, exploratory, and creative stages of software design [Petre, 2013]. Flexible modelling approaches focus on this creative aspect of modelling used for software design and aim to create collaborative tools that emulate whiteboard modelling, thus facilitating the creative process through informal modelling. Examples of such tools include Calico [Mangano et al., 2010] and Flexisketch [Wiieist et al., 2015]. Even though the models created by flexible modelling tools are inherently ambiguous and incomplete, it is possible to use them for certain lightweight analysis techniques. For example, partial models can be used to merge incomplete sketches created by multiple stakeholders in order to provide them with feedback about their composite model. Another example of lightweight analysis technique is described in [Motta et al., 2013], where analysis of timed properties of systems with stochastic behaviour using probabilistic model checking was deployed in Calico [Motta et al., 2013]. Informal modelling is one of the means that developers use to make sense of situations in which they encounter uncertainty. We intend to investigate the use of partial models as a back-end for informal modelling tools. The main challenge is to interpret informal sketches into partial models that capture the intuition of the modellers. Ideally, users of informal modelling tools should remain unaware of the partial modelling back-end, while still gaining meaningful feedback generated using partial modelling techniques.

Verification of semantic properties. In Section 4.1.1, we scoped partial model verification to syntactic properties in order to preserve the language independence of our verification approach. However,
certain semantic properties can be “unrolled” and then checked structurally using bounded model checking [Clarke et al., 2001]. Unrolling entails rewriting looping model structures up to a bounded number \(k\) of iterations. The resulting model structurally models those traces that iterate over the loop up to \(k\) times. It is thus possible to prove whether a semantic property holds within the given bound. Adapting this technique for partial models depends on correctly lifting the unrolling operation. We intend to use the technique for lifting graph-rewriting model transformations described in Chapter 5 as the basis for lifting unrolling. A potential challenge is the automatic generation of graph-rewriting rules for performing unrolling, given the semantics of a behavioural modelling language.

**Partial model execution.** In Section 7.3, we identified that one potential source of uncertainty in MDA is in the construction of model transformations. Since MDA transformations are also models, expressing and resolving uncertainty can be accomplished using the techniques described in Chapters 3 and 6, respectively. We noted, however, that in order for MDA practitioners to effectively practice decision deferral, we must also lift transformation execution, and pointed to work by [Strüßer et al., 2015] as a potential source of inspiration. This notion can be generalized to other kinds of partial models whose base language is executable. For example, since the execution of Petri nets can be simulated using graph rewriting [Ehrig and Padberg, 2004], we can envision the execution of partial Petri nets using lifted graph transformations. An obvious challenge is the state space explosion problem, as subsequent execution steps diffuse uncertainty throughout the initial Net. We intend to investigate under what conditions it is efficient to execute partial models, what kinds of reasoning can be performed with partial model execution traces, and what insights can be gained. One potential approach is to identify appropriate abstractions of the partial model’s Propositional Reduced Form (PRF) in order to leverage symbolic execution techniques.

**Improving partial model usability.** In [Famelis, M. and Santosa, S., 2013], we introduced a partial modelling notation that was intentionally designed based on D. Moody’s usability theory of notations [Moody, 2009]. This notation significantly influenced the creation of the “uncertainty tree” modelling sub-language of Mu-MMINT (which also had to take into account existing technological constraints). The uncertainty tree abstraction makes it possible to model uncertainty without having to use propositional logic. In addition, it provides a way to capture uncertain decisions as a first-class artifact. This way, the rationale for why a particular element is annotated as \texttt{Maybe} is also captured in the partial model. This, however, comes at a cost to expressiveness which is necessary for applying lifted transformations, since their output is not necessarily expressible using uncertainty trees. In the future, we intend to investigate striking a balance between usability and expressiveness. Feature modelling [Schobbens et al., 2006] is a potential source of inspiration, especially given that we have been able to successfully adapt transformation lifting for feature model-based annotative product lines [Salay et al., 2014].

**Uncertainty and Variability.** Variability management techniques have proliferated in industry to such a degree that it is now not uncommon for a company to maintain multiple product lines at once. The increasing complexity of features, variation points and domain artifacts requires more sophisticated ways of expressing and reasoning with questions such as “If we include this feature, will all products satisfy that property?”, “What are the implications of mapping this feature to that piece of code?”. In such contexts, the synergy between uncertainty and variability modelling (essentially, the combination
of existential and universal quantification over power-sets of possibilities) can provide a powerful way to articulate and reason with complicated business scenarios. In current work [Chechik et al., 2016], we are developing a technique for performing verification of properties on product lines that contain points of uncertainty, as well as a methodology for responding to verification results.

Similar synergies arise in the field of dynamically adaptive systems (DASs) [Esfahani and Malek, 2012]. In DASs, design decisions are made at run-time to adapt to changes in the environment in which the software system operates. In fact, DASs are often understood as dynamic product lines [Bencomo et al., 2008]; thus many of the synergies between uncertainty and variability described above are also relevant to DAS engineering. In [Ramirez et al., 2012], several sources of design-time uncertainty are identified for DASs for which no mitigation strategy currently exists. The techniques presented in this thesis may be directly applicable in addressing this gap. For example, decision deferral can be used to mitigate uncertainty caused by trying to avoid making “design decisions based on misguided and subjective preference” [Ramirez et al., 2012].

**Lifting industrial-grade transformation languages.** The approach for lifting graph-rewriting transformations presented in Chapter 5 demonstrates the feasibility of simultaneously applying a graph rewriting rule to an entire set of concretizations. However, industrial-grade model transformation requires languages that are orders of magnitude more complex, with constructs such as control, iteration, data structures, etc. It is therefore necessary to go beyond simple graph-rewriting. The experience gained from lifting transformations for partial models has allowed us to adapt this technique to software product lines [Salay et al., 2014]. Subsequently, we were able to use the lifting of individual graph-rewrite rules in order to lift parts of DSLTrans. DSLTrans is a full-fledged model transformation language that combines graph-rewriting with advanced language constructs and is rich enough to implement real-world transformations [Lúcio et al., 2010]. Using the lifted version of the DSLTrans engine, we were able to execute an industrial-grade model transformation of product lines from the automotive domain [Famelis et al., 2015b]. In current work, we are expanding lifting to the entire DSLTrans language. We intend to use the insights and experience gained from this, to lift widely used transformation languages such as ATL [Jouault et al., 2006].
Bibliography

URL http://doi.acm.org/10.1145/38713.38724

URL http://doi.acm.org/10.1145/320107.320112


Ambler, S. W., 2005: The Elements of UML (TM) 2.0 Style. Cambridge University Press.

URL http://dx.doi.org/10.1007/978-3-540-75209-7_30


URL http://dx.doi.org/10.1023/A%3A1011276507260


URL http://www.eclipse.org/cdo/


URL http://wiki.eclipse.org/Teneo/

URL http://dx.doi.org/10.1007/3-540-56689-9_43


URL http://dx.doi.org/10.1007/s00766-013-0198-z


URL http://dx.doi.org/10.1007/978-3-642-16145-2_22


URL http://dx.doi.org/10.1007/978-3-540-30494-4_20


URL http://doi.acm.org/10.1145/2076450.2076469


URL http://dx.doi.org/10.1007/BFb0046842


URL http://dx.doi.org/10.1007/11768869_14


URL http://dx.doi.org/10.1007/978-3-642-21292-5_3


URL http://dx.doi.org/10.1007/978-3-319-06410-9_26

URL http://drops.dagstuhl.de/opus/volltexte/2005/23


URL http://doi.acm.org/10.1145/1858996.1859003


URL http://doi.acm.org.myaccess.library.utoronto.ca/10.1145/1060710.1060712


URL http://www.omg.org/spec/UML/2.5

URL http://www.omg.org/spec/XMI/2.5.1

URL http://dl.acm.org/citation.cfm?id=2486788.2486883


URL http://dl.acm.org/citation.cfm?id=2486788.2486883


URL http://doi.acm.org.myaccess.library.utoronto.ca/10.1145/1164394.1164418

URL http://dx.doi.org/10.1016/j.scico.2009.02.007

URL http://dx.doi.org/10.1007/978-3-642-36654-3_2


URL http://dx.doi.org/10.1007/s00766-013-0170-y


URL http://dx.doi.org/10.1007/978-3-642-37057-1_3


URL http://dx.doi.org/10.1007/s00450-012-0233-1


URL http://doi.acm.org/10.1145/1093382.1093385


URL http://dx.doi.org/10.1007/s11245-006-0021-2
Appendices
Appendix A

Proofs

A.1 Representing Uncertainty

Proof of Theorem 3.2:

Proof. If the corresponding variable of the Maybe-annotated atom $a$ is in the backbone ($\phi \models \text{atomToProposition}(a)$), then replacing $\text{atomToProposition}(a)$ by $\text{True}$ in $\phi$ produces a logically equivalent formula and so has no impact on the set of concretizations $C(M)$. Furthermore, annotating $a$ with True has no effect on $C(M)$ since $a$ must already occur in all concretizations as $\phi \models a$. Similarly, if $\neg \text{atomToProposition}(a)$ is in the backbone, replacing $\text{atomToProposition}(a)$ by False has no impact on $C(M)$, while removing it also has no impact as no concretization contains it. Therefore, since the algorithm has no effect on $C(M)$, it must be that $M \sim M^{\text{GRF}}$. \hfill \square

Proof of Theorem 3.3:

Proof. Assume that such a model $M'$ exists. Then there exists some atom $a$ which is annotated with Maybe in $M^{\text{GRF}}$. Yet in $M'$, it is either annotated with True (i.e., all of its concretizations have it) or is absent altogether. For $a$ to be Maybe-annotated in $M^{\text{GRF}}$, it must be the case that $\text{atomToProposition}(a)$ is not in the backbone of the May formula of $M$ (Lines 6, 9). Therefore, $M$ has some concretization $m_1$ that contains $a$ and some other concretization $m_2$ that does not. In other words, $M$ has more concretizations than $M'$, which is a contradiction. \hfill \square

Proof of Theorem 3.4:

Proof. An atom $a$ annotated with True must occur in every concretization in $C(M)$. Thus, annotating it with Maybe and conjoining $\text{atomToProposition}(a)$ with the May formula $\phi$ has no affect on $C(M)$ because $\phi \land \text{atomToProposition}(a) \models \text{atomToProposition}(a)$, i.e., $a$ is exists in all concretizations. Therefore, it must be that $M \sim M^{\text{PRF}}$. By definition, every atom of $M$ is either annotated with True or with Maybe. On Line 5, every True-annotated atom is annotated with Maybe. Therefore all elements of $M^{\text{PRF}}$ are annotated with Maybe. \hfill \square

Proof of Theorem 3.5:
Appendix A. Proofs

Proof. We show that a model $m \in A$ iff $m \in C(M)$.

Forward Direction: If $m \in A$ then $\text{graphToPropositionSet}(m) \subseteq \text{graphToPropositionSet}(G^T_u)$ since $G^T_u = \text{GraphUnion}(A)$. Furthermore, all atoms in $G^T_u$ that are not inside $m$ must be annotated with Maybe due to Lines 4-5, because they are not common to all concretizations in $A$ (Line 2). Thus, $m$ is concretization of $G$ because there is a disjunct (constructed as $\phi_m$) in the May formula $\phi$ corresponding to $m$ that sets all Maybe atoms outside of $m$ in $G$ to False and all Maybe atoms inside of $m$ in $G$ to True.

Reverse Direction: By Line 12, each disjunct $\phi_m$ of the May formula $\phi$ sets all the Maybe-annotated atoms in $G^T_u$ to either True or False and thus defines a concretization $m$ of $M$. Furthermore, there can be no other concretizations of $M$, since such a concretization would have to satisfy $\phi$. This holds because: (a) $\phi$ has exactly one disjunct for each model in $A$ since it is constructed this way in Lines 11-14. (b) Since $\phi$ is expressed as a disjunction of conjuncts of all Maybe-annotated atoms, only the concretizations corresponding to the disjuncts satisfy it. Therefore, if $m \in C(M)$, then $m \in A$. \hfill \Box

Proof of Theorem 3.6:

Proof. To prove that $M$ is in GRF, we must show that it contains the least possible number of models of Maybe-annotated atoms for partial models with the set of concretizations $A$. Assume that there exists a model $M'$ such that $M \sim M'$ but $M'$ contains fewer Maybe-annotated atoms. Then there must exist some atom $a$ which is annotated with Maybe in $M$ but with True in $M'$. The only way for $a$ to have been annotated with Maybe is if there exists some concretization $m$ in $A$ such that $a \notin m$ (cf. Line 2 of Algorithm 3). However, if $a$ is annotated with True in $M'$ and since $C(M') = C(M) = A$, that means that $\forall m \in A \cdot a \in m$, which is a contradiction. \hfill \Box

A.2 Resolving Uncertainty

Proof of Theorem 6.1:

Proof. Let $M' = \text{Keep}(M,e)$. To prove that $M' \preceq M$, we put both in PRF. Since by definition $\phi' = \phi[\text{True} / \text{atomToProposition}(e)]$, it follows that $\phi' \Rightarrow \phi$. Since each concretization is a valuation of the May formula, $C(M') \subseteq C(M)$, i.e., $M' \preceq M$. The proof for $\text{Drop}$ is identical. \hfill \Box

Proof of Theorem 6.2:

Proof. As the resulting model is constructed by the formula $\phi_M^{\text{PRF}} \land \sim \phi_p$, its set of concretizations is exactly the subset of concretizations of the original partial model for which the property was violated. In other words, $M \neg \preceq p \preceq M$. \hfill \Box
Appendix B

Generation of Random Inputs for Experimentation

Here, we provide additional details about the randomly generated inputs used for the experiments described in Sections 4.3 and 6.2.

We illustrate our encoding using a toy randomly generated model $M^r$ shown in Figure B.1(b). $M^r$ is an untyped graph with two nodes and three edges. Its accompanying May formula specifies the allowable configurations of its Maybe-annotated atoms ($M^r$ is not in GRF). The corresponding SMT-LIB encoding is shown in Figure B.1(a).

The SMT-LIB encoding of the model follows the semantics of the naming conventions for propositional variables described in Section 2.1.2. For example, the expression $\exists n_2 \in M^r \cdot \text{type}(n_2) = \text{Node}$ is captured by the assertion in Line 24. More specifically, for each type $t$, we create a corresponding boolean function $f_t$. Each $t$-atom is assigned an ordinal integer $i$. The existence of the $i$-th $t$-atom is expressed by the assertion $f_t(i) = \text{True}$. For Maybe-annotated elements, the corresponding assertions are commented out (e.g., Line 23 for node $n_1$ and Line 31 for edge $e_3$).

We also encode the source and target nodes for every edge using our naming convention. For example, the assertions in Lines 26-27 capture the expression $s(e_1) = n_1 \land t(e_1) = n_2$.

Lines 35-39 encode the May formula of $M^r$ as a disjunction of conjunctions. We generate May formulas from the EMF representation of the generated graph. Each disjunctive clause contains all of the Maybe-annotated elements of $M^r$, a random subset of which is negated, taking care to only create clauses that correspond to valid graphs. For example, we do not generate a disjunctive clause that would include the assertion $\neg n_1 \land e_1$, as it would violate the graph well-formedness constraints. In our example, the May formula mandates that $M^r$ has exactly the two concretizations shown in Figure B.1(c-d).

For each property, we generate the corresponding propositional grounding. In our example, we have grounded the “Multiple Inheritance”-inspired property (Property 1) from Table 4.4, which mandates that “a node can have at most one outgoing edge”. Since this must hold for all nodes, this is a universal property, and we ground it to a conjunction of sub-clauses, one for each node. In our example, $M^r$ contains two nodes, so the conjunction in Line 42 has two sub-clauses: for $n_1$ (Lines 43-45) and for $n_2$ (Line 47).

Each property is grounded using a specialized procedure. For Property 1, we do the following: if a node has at most one outgoing edge, we create a sub-clause that is vacuously true, such as for $n_2$ in
Figure B.1: (a) SMT-LIB encoding of a randomly generated partial model $M_r$. (b) The partial model $M_r$. (c-d) Concretizations of $M_r$. (e) Configurations of the outgoing edges of node 1 that satisfy Property 1. Concretization (c) satisfies Property 1 whereas concretization (d) does not.
Line 47. If it has more than one edge, we create a disjunction which effectively expresses the fact that only one of the edges can exist in a concretization. For \( n_1 \), this implies three possibilities for edges \( e_1 \) and \( e_3 \). These are depicted graphically in Figure B.1(e) and encoded in Lines 44-46.

Checking Property 1 on \( M^r \) involves performing two checks: one with the original property (as shown in Figure B.1(a)) and the other – with the negation of this property. Each is done by calling Z3. In this example, both Property 1 and its negation are satisfiable: the former because of the concretization in Figure B.1(c), and the latter – because of the one in Figure B.1(d). Based on Table 4.1, the result of checking Property 1 on \( M^r \) is Maybe.

For performing experiments with sets of classical models (E2), we encoded each classical model \( m \) as a formula \( \phi_m = \bigwedge v_i \), where \( v_i \in \{\text{graphToPropositionSet}(m)\} \). We grounded each property \( p \) over the set \( \text{graphToPropositionSet}(m) \) in the same way as for partial models to get the corresponding formula \( \phi_p \). Then for each model, we constructed a formula of the form \( \phi_m \land \phi_p \). The property holds iff this formula is satisfiable.
Appendix C

ORM Benchmark

In this appendix, we provide additional details about the Object-Relational Mapping (ORM) in Chapter 5.

Figure C.1 shows the simplified class diagram metamodel used in the benchmark.

Figure C.2 shows the metamodel of relational database schemas used in the benchmark.

Figure C.3 shows the first three Henshin rules classToTable, associationToTable, and attributeToColumn used to perform the ORM transformation.

Figure C.4 shows the last two Henshin rules attributeToForeignKey and associationToForeignKey used to perform the ORM transformation.

Figure C.5 shows the diagram of the partial Ecore metamodel with the largest set of concretizations that was used as input for the ORM benchmark.

The May formula of this partial model, expressed in SMT-LIB is shown in Figure C.6.
Figure C.1: Simplified class diagram metamodel.
Figure C.2: Relational database schema metamodel.
Figure C.3: Rules `classToTable`, `associationToTable`, and `attributeToColumn` used to perform the ORM transformation.
Figure C.4: Rules `attributeToForeignKey` and `associationToForeignKey` used to perform the ORM transformation.
Figure C.5: Partial Ecore metamodel, used as input to the ORM benchmark.
; Legend
; a: association eParameters
; b: class EParameter
; c: attribute instance
; d: attribute value
; e: association eOpposite
; f: attribute resolveProxies
; i: association eFactoryInstance
; j: attribute nsURI
; k: association eSuperTypes
; l: association eSuperPackage
; m: attribute containment
; n: attribute container
(and
  (= a b)
  (= c d)
  (= e f)
  (= i j)
  (= k l)
  (= m n)
)
Appendix D

Supplementary Material to Worked-Out Examples

In this appendix, we provide additional details about the worked examples in Chapter 8.

D.1 UMLet Bug #10

Figure D.1 shows the Mu-MMINT model K0 that encodes the UMLet model shown in Figure 8.1. The encoded model contains the relevant slices of both the class diagram and the sequence diagram (bottom), as well as traceability links between them, linking messages in the sequence diagram to operations in the class diagram and objects to their classes. In the sequence diagram model, objects and lifelines are represented by the same model element. Mu-MMINT uses yellow and pink stars to respectively indicate edges that represent the source and target lifelines of messages.

Table D.1 shows the valid combinations of strategies for repairing the ClasslessInstance and DanglingOperation consistency violations. Each combination represents a repair of the model shown in D.1, i.e., a concretization of the partial model K1, shown in Figure D.2.

Figure D.2 shows the diagram of the partial model K1 created by the maintainer to express her uncertainty about which of the 44 alternative repairs in Table D.1 to select. To enhance diagram readability, we have hidden the labels of the arrow model elements. This has resulted in hiding Maybe annotations as well; however, it is easy to deduce that edges that are in K1 but not in K0 have Maybe annotations.

Figure D.3 shows the ground propositional encoding $\phi_{U1}$ of the property U1, expressed in SMT-LIB [Barrett et al., 2010]. The formula encodes the property as a conjunction of the variables (cf. atomToProposition) of the messages that are required to perform pasting.

Figure D.4 shows the ground propositional encoding in SMT-LIB of the sequence diagram well formedness constraints for the slice of K1 involved in checking U1. Specifically, we check that for each message in Figure D.3, there is an operation reference that maps it to the appropriate method definition. The abbreviation “or” in the name of the variables means “operation reference”.

Figure D.5 shows the ground propositional encoding $\phi_{U2}$ of the property U2 for K1, in SMT-LIB. Specifically, we check whether the message setComponentZOrder exists in every concretization. We have
combined the property with the sequence diagram well-formedness constraint additionally requiring the appropriate operation reference to the method definition.

Figure D.6 shows the model K2, a concretization of K1. It was created by invoking the operator \texttt{GenerateCounterExample} with inputs K1 and U1. Mu-MMINT has greyed out \texttt{Maybe} elements of K1 that are not also part of K2.

Figure D.7 shows the diagram of the partial model K3, resulting from invoking the operator \texttt{Constrain} with inputs K1 and U1. To enhance diagram readability, we have hidden the labels of edge elements. Every edge that is outgoing from the elements \texttt{new}, \texttt{moveToTop}, \texttt{setComponentZOrder}, and \texttt{positioner} is annotated with \texttt{Maybe}.

Figure D.8 shows the (concrete) model K4, resulting from invoking the operator \texttt{Decide} with K3 as input while choosing the repairs RC4 and RD2.
Figure D.1: MU-MMINT model K0.
Table D.1: Valid combinations of repair strategies for ClasslessInstance and DanglingOperation.

<table>
<thead>
<tr>
<th>#id</th>
<th>ClasslessInstance</th>
<th>DanglingOperation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RC1</td>
<td>RD3</td>
</tr>
<tr>
<td>2</td>
<td>RC2(self)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(Paste)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RC2(e)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(GridElement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(getX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(getY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>RC2(handler)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(DiagramHandler)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(mult)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>RC2(pnl)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(DrawPanel)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(setComponentZOrder)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>RC2(AddElement)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(AddElement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(execute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>RC3(Paste)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(Paste)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>RC3(GridElement)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(GridElement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(getX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(getY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>RC3(DiagramHandler)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(DiagramHandler)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(mult)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>RC3(DrawPanel)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(DrawPanel)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(setComponentZOrder)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>RC3(AddElement)</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(AddElement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(execute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>RC4</td>
<td>RD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RD2(NewClass)</td>
</tr>
<tr>
<td>43</td>
<td></td>
<td>RD3</td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure D.2: Diagram of the Mu-MMINT partial model K1.
Figure D.3: Ground propositional encoding of the property U1.

Figure D.4: Ground propositional encoding of sequence diagram well-formedness constraints involved in checking the property U1.

Figure D.5: Ground propositional encoding of the property U2.
Figure D.6: Concretization K2 of the partial model K1, a counterexample demonstrating why checking U1 yields Maybe. MU-MMINT has greyed out elements of K1 that are not also part of K2.
Figure D.7: Diagram of the partial model K3.
Figure D.8: Diagram of the final model K4, implementing the repairs RC4 and RD2.
Appendix D. Supplementary Material to Worked-Out Examples

D.2 Petri net metamodel

Figure D.9 shows the MU-MMINT diagram of the partial metamodel N0. Meta-associations are decorated with the icon “✓”, whereas containment references — with the icon “✓”. Maybe elements are annotated with “[M]” and one or more alternatives from the uncertainty tree of N0 in square brackets. The uncertainty tree of N0 is shown in Figure D.10.

The May formula of N0 is constructed from the uncertainty tree using the technique described in Section 7.4. Specifically:

- The May formula is a conjunction of the decision variables:
  \[ d_1\text{ArcClasses} \land d_4\text{Locations} \land d_5\text{TokenClass} \land d_7\text{Executions} \]

- Each decision variable is equivalent to an exclusive disjunction of the alternative variables. For example:
  \[ d_1\text{ArcClasses} \Leftrightarrow (d_1\text{ynw} \oplus d_1\text{yw}) \oplus d_1\text{n} \]

- Each alternative variable is equivalent to the conjunction of the Maybe elements that are annotated with the alternative and the negations of the Maybe elements that are annotated with other alternatives of the same decision. For example:
  \[ d_1\text{n} \Leftrightarrow \text{src}\text{.placeToTransition}\text{Association} \land \text{src}\text{.transitionToPlace}\text{Association} \land \text{dst}\text{.placeToTransition}\text{Association} \land \text{dst}\text{.transitionToPlace}\text{Association} \land \neg \text{PlaceToTransition}\text{Class} \land \neg \text{TransitionToPlace}\text{Class} \land \ldots \land \neg \text{weight}\text{P2T}\text{Attribute} \land \neg \text{getElement}\text{P2T}\text{Operation} \land \neg \text{setWeight}\text{P2T}\text{Operation} \land \ldots \]

With the exception of the class Location, the various meta-attributes, and the getter and setter operators, the diagram of the metamodel N0 was created by merging slices of the AMZ metamodels listed in Table 8.3. We sliced the AMZ metamodels in order to get a model that can be used with the ORM transformation described in Appendix C, which requires the input class diagram to have a flat class inheritance hierarchy.

Figure D.11 shows the ground propositional encoding \( \phi_{U3} \) of the property U3, expressed in SMT-LIB [Barrett et al., 2010]. The formula encodes the property as a conjunction of the variables (cf. atomToProposition) of the tables that are needed to map tuples from tables representing graphical PTN elements to tuples of the table Location.

Figure D.12 shows the partial PTN metamodel N2 that results from invoking the operator Decide on the partial model N0 to select the alternative \( d_4\text{y} \) of the decision \( d_4\text{Locations} \) in the uncertainty tree in Figure D.10.

Figure D.13 shows the diagram of the partial PTN metamodel N3 that results from invoking the operator Expand on the partial model N2 to include uncertainty about which domain-specific PTN constructs should be included in the CONCMOD tool.

Figure D.15 shows the concrete PTN metamodel N5, resulting from invoking the operator Decide with N3 as input and making the decisions \( d_1\text{yw}, d_2\text{n}, d_3\text{n}, d_5\text{y}, d_6\text{n}, d_7\text{y}, d_8\text{n}, d_9\text{n}, d_10\text{n} \) from the uncertainty tree in Figure D.14.
Figure D.9: Diagram of the partial metamodel N0.
**d1_ArcClasses** Are arcs represented as separate meta-classes?

[**d1ynw**] Yes. Arcs have weights.
[**d1yw**] Yes. Arcs do not have weights.
[**d1n**] No.

**d4_Locations** Is the location of elements in the diagram stored?

[**d4y**] Yes.
[**d4n**] No.

**d5_TokenClass** Is there a separate meta-class for tokens?

[**d5y**] Yes.
[**d5n**] No.

**d7_Executions** Is there a mechanism for representing executions?

[**d7y**] Yes.
[**d7n**] No.

Figure D.10: Uncertainty tree of the partial metamodel N0.

```
(and
  (node placeToLocation)
  (node transitionToLocation)
  (node tokenToLocation)
  (node placeToTransitionArcToLocation)
  (node TransitionToPlaceArcToLocation)
)
```

Figure D.11: Ground propositional encoding of the property U3.
Appendix D. Supplementary Material to Worked-Out Examples

**d1. ArcClasses** Are arcs represented as separate meta-classes?

[d1yw] Yes. Arcs have weights.
[d1yn] Yes. Arcs do not have weights.
[d1n] No.

**d5. TokenClass** Is there a separate meta-class for tokens?

[d5y] Yes.
[d5n] No.

**d7. Executions** Is there a mechanism for representing executions?

[d7y] Yes.
[d7n] No.

---

Figure D.12: Partial PTN metamodel N2.
Figure D.13: Partial PTN metamodel N3. The uncertainty tree is shown in Figure D.14.
d1_ArcClasses Are arcs represented as separate meta-classes?

[d1ynw] Yes. Arcs have weights.
[d1yw] Yes. Arcs do not have weights.
[d1n] No.

d2_ArcKinds Are there different kinds of arcs?

[d2y] Yes.
[d2n] No.

d3_Priority Do transitions have priority?

[d3y] Yes.
[d3n] No.

d5_TokenClass Is there a separate meta-class for tokens?

[d5y] Yes.
[d5n] No.

d6_Timed Are transitions timed?

[d6y] Yes.
[d6n] No.

d7_Executions Is there a mechanism fore representing executions?

[d7y] Yes.
[d7n] No.

d8_ColouredTokens Are tokens coloured?

[d8y] Yes.
[d8n] No.

d9_Stochastic Are transitions stochastic?

[d9y] Yes.
[d9n] No.

d10_Guards Are arcs guarded?

[d10y] Yes.
[d10n] No.

Figure D.14: Uncertainty tree of the partial metamodel N3.
Figure D.15: Diagram of the final PTN metamodel N5.
Index

Agile, 6
Aleatory uncertainty, 3
Applicability condition, 26
  lifted, 68
Architectural layer, 29
Articulation stage, 84
  operators, 87
Atom, 21
atomToProposition, 21
Backbone, 42
Base language, 35
Benchmark, 71
Bug #10, 102

CDO, 71
CIM, 29
Clafer, 47
Classical, 27
Cloning, 6
Computation Independent Model, 29
Concretization, 13, 32
Confluence, 28, 71
Consistent partial model, 32
Constrain operator, 90
Construct operator, 87
Correctness criterion, 66
Counterexample, 53

Databases, 61
Decide operator, 90
Deconstruct operator, 89
Deferral stage, 84
  operators, 88
Degree of uncertainty, 32, 74
Design space exploration, 99
Diagnosis, 50

Diagnostic core, 54, 78
Donald Rumsfeld, 8
Drop operator, 75
Ecore, 71
Encapsulate Variable, 64
Epistemic uncertainty, 3
Equivalent partial model, 40
Expand operator, 87
F+ and F-, 52
Feature model, 47
FOL, 24
Forking, 6

GenerateCounterExample operator, 90
GenerateDiagnosticCore operator, 90
GenerateExample operator, 90
Graph, 20
  Typed graph, 21
Graph rewriting, 25
Graph rewriting system, 27
Graph transformation, 25
Graph union, 23
Graphical Reduced Form, 42
  Efficiency and sensitivity of construction, 78
graphToPropositionSet, 22
GRF, 42
Grounded formula, 24

Henshin, 10, 71

IDE, 96
Irreducible uncertainty, 3

Keep operator, 75
Known unknowns, 8

Lean, 6
Left hand side, 25
LHS, 25
Lifted rule applicability, 68
Lifted rule application, 69
Lifting, 66
correctness criterion, 66
MakePartial operator, 87
Manual decision making, 75
Matching site, 25
MAVO, 47
May formula, 13, 32
Maybe elements, 13, 32
MDA, 28, 92
Megamodel, 81, 96
Metamodel, 20, 21, 45
constraining, 81
partialized, 36
reasoning, 60
MMINT, 71, 98
Model, 20, 21
Model Driven Architecture, 28, 92
Model management, 86
Model transformation, 25
MU-MMINT, 96
NAC, 25
Negative application condition, 25
Nondeterminism, 9, 46

Object-Relational Mapping, 71
OCL, 24
OMG, 28
ORM, 71

Partial behavioural model, 45
Partial behavioural models
reasoning, 60
Partial model, 32
consistent, 32
construction, 44
equivalent, 40
notation, 33
reduced form, 41
refinement, 33
semantics, 33
singleton, 32
transient, 86
Partialization, 36
Physics of Notations, 38
PIM, 29
Platform, 29
Platform Independent Model, 29
Platform Specific Model, 29
Presence conditions, 47
PRF, 43
Programming by Optimization, 6
Properties, 24
Property checking, 52
Property-driven refinement, 77
Propositional Reduced Form, 43
 Provisionality, 6
PSM, 29
PtPP example, 1
Reduced form, 41
Reducible uncertainty, 3
Refinement, 33, 51, 75
property-driven, 77
Removal/addition of behaviour, 81
Resolution stage, 84
operators, 89
RHS, 25
Right hand side, 25
Rule, 25
applicability condition, 26
application, 25, 27
Lifted applicability, 68
Lifted application, 69
parts, 25
Scope, 7
scope, 46
Semantic properties, 24
Singleton partial model, 32
Software produce lines, 46
reasoning, 61
Software product lines
annotative, 47
staged configuration, 81
SolvEx example, 63
Span, 7
span, 46
SPL, 46
Syntactic properties, 24
Teneo, 71
Termination, 28, 70
Traces, 51
Transform operator, 88
Transformation
  application, 25
  uncertainty-reducing, 76
Transformation rule, 25
Transient, 86
UMlet, 102
Uncertainty, 2
  Aleatory, 3
  degree, 74
  Design-time uncertainty, 3
  Environmental uncertainty, 3
  Epistemic, 3
  Irreducible, 3
  propagation, 95
  Reducible, 3
  removal, 75
  resolution, 75
Uncertainty management operator, 87
Uncertainty tree, 96
Uncertainty Waterfall, 83
  extended, 85
  operators, 87
  transitions, 85
Underspecification, 9, 46
Unknown unknowns, 8
URT, 76
Variability, 47
  management, 100
  reasoning, 61
  variability-aware techniques, 61
Verification, 24, 50
feasibility, 55
scalability, 55
semantics, 51
Verify operator, 88
Waterfall model, 85
Well-formedness constraints, 21
WFC, 21