Université de Montréal

Articulating design-time uncertainty with Druide

par

Mouna Dhaouadi

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Intelligence Artificielle

September 18, 2020

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Articulating design-time uncertainty with DRUIDE

présenté par

Mouna Dhaouadi

a été évalué par un jury composé des personnes suivantes :

Dr. Eugene Syriani

(président-rapporteur)

Dr. Michalis Famelis

(directeur de recherche)

Dr. Houari Sahraoui

(membre du jury)

Résumé

Les modélisateurs rencontrent souvent des incertitudes sur la manière de concevoir un modèle logiciel particulier. Les recherches existantes ont montré comment les modélisateurs peuvent travailler en présence de ce type d' "incertitude au moment de la conception". Cependant, le processus par lequel les développeurs en viennent à exprimer leurs incertitudes reste flou.

Dans cette thèse, nous prenons des pas pour combler cette lacune en proposant de créer un langage de modélisation d'incertitude et une approche pour articuler l'incertitude au moment de la conception. Nous illustrons notre proposition sur un exemple et l'évaluons non seulement sur deux scénarios d'ingénierie logicielle, mais aussi sur une étude de cas réel basée sur les incertitudes causées par la pandémie COVID-19. Nous menons également un questionnaire post-étude avec les chercheurs qui ont participé à l'étude de cas. Afin de prouver la faisabilité de notre approche, nous fournissons deux outils et les discutons. Enfin, nous soulignons les avantages et discutons des limites de notre travail actuel.

Mots clés: Langage basé sur l'incertitude, langage de modélisation, ingénierie basée sur les modèles, décision au moment de la conception, méga-modélisation.

Abstract

Modellers often encounter uncertainty about how to design a particular software model. Existing research has shown how modellers can work in the presence of this type of "design-time uncertainty". However, the process by which developers come to elicit and express their uncertainties remains unclear.

In this thesis, we take steps to address this gap by proposing to create an uncertainty modelling language and an approach for articulating design-time uncertainty. We illustrate our proposal on a worked example and evaluate it not only on two software engineering scenarios, but also on a real case study based on uncertainties caused by the COVID-19 pandemic. We also conduct a post-study questionnaire with the researchers who participated in the case study. In order to prove the feasibility of our approach, we provide two tool supports and discuss them. Finally, we highlight the benefits and discuss the limitations of our current work.

Keywords: Uncertainty-wise language, modelling language, model-driven engineering, design-time decision, mega-modelling.

Contents

List of tables

List of figures

List of Abbreviations

Druide	Design and Requirements Uncertainty Integrated Development Editor
DeTUM	Design Time Uncertainty Management Model
PTNs	Petri Nets
SMM	Software Model Management
DMN	Decision Model and Notation
DRD	Decision Requirements Diagram
DOPLER	Decision-Oriented Product Line Engineering for effective Reuse
OCL	Object Constraint Language
MSR	Mining Software Repositories
SPLE	Software Product Line Engineering

Chapter 1

Introduction

1.1. Research context

Making decisions at the right time is a key factor to successful software engineering tasks. Making decisions prematurely, in the absence of all necessary information, may lead to losses in terms of wasted effort, as well as to potentially dangerous unwarranted assumptions. In real life, developers are often faced with uncertainty about various decisions. Thus, researchers have investigated ways to provide them with the ability to work in the presence of uncertainty, until the most opportune moment comes. In general, existing languages focus on representing the uncertainty in the models. In this thesis, we focus on the uncertainty that concerns the design of the models, also called 'design-time uncertainty' [**15**].

Several researchers have attempted to model and manage design-time uncertainty. These uncertainty-aware software development methodologies include the DETUM approach that uses partial models [**15**] to perform engineering tasks such as reasoning [**12**], refinement [**33**], and transformations [**16**], while leveraging a set of possible alternative designs. Although partial models deal uniquely with one type of partiality, other partiality types have been proposed in the literature in the context of the MAVO framework [**30**]. The U-Model [**45**] is a conceptual model for uncertainty that has been proposed specifically for Cyber-Physical Systems. The U-RUCM [**46**], an extension of the RUCM [**43**] methodology, bases on the U-Model in the aim of specifying uncertainty for use case modelling. Other approaches for modelling design-time uncertainty have been developed in the context of bidirectional transformations using JTL [**10**], architectural interfaces [**42, 18**], and pattern matching [**36**].

Besides, design-time uncertainty concerns modelling uncertainty about decisions. Thus, it is closely related to uncertainty about how to make decisions in a given problem space. Approaches for modelling decisions include the OMG's Decision Model and Notation (DMN) [**27**] and the DOPLER Variability Modelling Language [**6**].

1.2. Problem statement

Existing research has shown how modellers can work in the presence of design-time uncertainty. However, to the best of our knowledge, there is no contribution that addresses the process by which developers can express their design uncertainties and then, come to evolve and articulate the design decisions. Thus, this uncertainty articulation process remains unclear, which makes it hard to adopt techniques for managing design-time uncertainty in the regular rhythm of modelling, which in turn hinders their applicability.

Besides, due to the lack of adequate support and tools, currently when a modeller is faced with some uncertainty, he has to stop modelling until further information is available. In order to express his design-time uncertainties, to elicit his design decisions or to keep track of their evolution, the modeller has to use other artifacts such as papers or textual notebooks, misuse a feature of his modelling tool or use other modelling tools.

In this thesis, we propose to fill this gap in the literature by creating a modelling language, an approach and tool support for the articulation of uncertainty.

1.3. Proposed research contributions

The research contributions of this thesis are as follows:

- Proposing an uncertainty modelling language that can be used with different modelling artifacts.
- Proposing a set of constraints that assure the well-formedness and consistency of the resulting uncertainty model.
- Proposing an operator-based methodology for the articulation of uncertainty.
- Defining a set of constraints that regulate the use of the operators and check their correct usage.

In addition, we:

- Illustrate our language and approach on a lab-based software engineering example.
- Evaluate our language and approach not only on two worked software engineering examples taken from the literature, but also on a real-case scenario based on the COVID-19 pandemic. We also conduct a post-study questionnaire with the three researchers who participated in the case study.
- Show the practicality of our proposal by providing two alternatives for tool support, and discuss them.
- Discuss our work and develop the outline of our future research agenda.

1.4. Running Example

In this section, we present an example that we will use throughout this thesis to illustrate our language and approach. The example has been initially presented in [**15**].

In this example, the modeller wants to create a tool called CONCMOD, for modelling concurrent systems. The tool uses the widely-used formalism of Petri Nets (PTNs) [**23**]. The modeller creates an initial draft of a meta-model for developing PTNs, presented in Figure 1.1. The meta-model contains three classes. The class "Net" represents the entire Petri Net. It contains a set of transitions and places. These are represented by two more classes: the class "Transition" represents the events or actions that may occur in the system, and the class "Place" represents the conditions that need to be met in order for an event to happen. Each class has an attribute "name" that serves as a unique identifier. However, the modeller is faced with some design uncertainty about how to represent other elements of the meta-model, such as arcs and tokens. Specifically, she finds out that several variant PTNs meta-models exist [**1**], and she does not know what elements of which ones are relevant for CONCMOD. We will elaborate on this uncertainty in detail later in section 3.3.3.

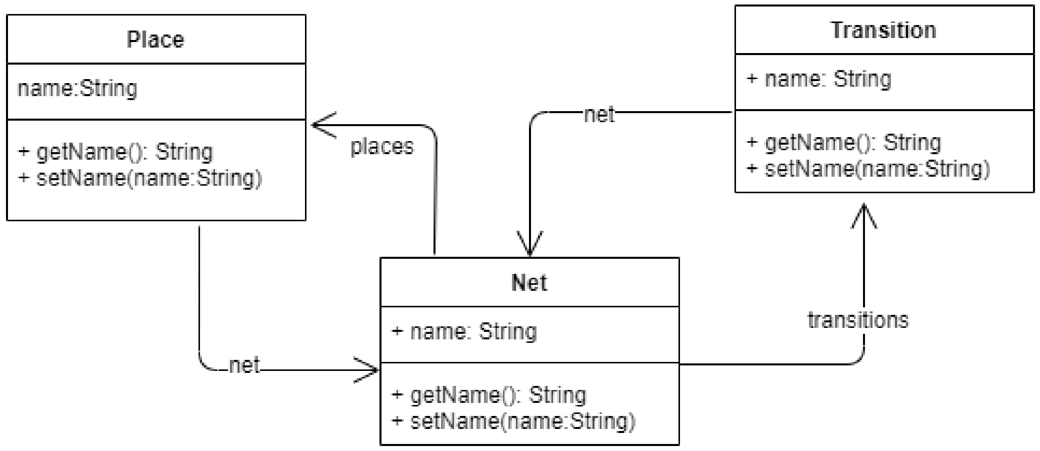


Fig. 1.1. The part of the PTNs meta-model that don't contain uncertainty

1.5. Thesis outline

This thesis is structured as follows. First, we introduce all necessary background knowledge and related work in Chapter 2. Then, we introduce our language and methodology and illustrate their usage in Chapter 3. We evaluate our work on the literature-based examples in Chapter 4 and on the real-world case study in Chapter 5. After that, we present our findings on providing tool support for our work in Chapter 6, and we discuss our proposal in Chapter 7. Finally, we conclude and develop our future contributions in Chapter 8.

Chapter 2

Background and Related work

In this chapter, we present all necessary background to introduce our work. We start by introducing mega-models and model management. Then, we present related work on uncertainty modelling and we explore in details the aspects that we build our proposal upon. Finally, we introduce some approaches for decision-oriented modelling.

2.1. Model management and megamodels

Modern software development requires managing multiple diverse software artifacts [**31**]. Thus, several approaches have been proposed in the literature for the composition of heterogeneous modelling languages. In [**9**], the authors outline three existing techniques for the reuse and composition of meta-models (meta-model merge, meta-model interfacing and class refinement) and propose a new meta-model composition technique called *Template Instantiation*. Haber et al. [**20**] propose a syntax-level integration approach of textual languages.

The interactions of heterogeneous languages within a software system poses several problems [**24, 38, 4**]. Model management refers to the challenges caused by working simultaneously with a collection of heterogeneous models. The need for model management has first emerged in the area of meta-data management [**2**], then it was introduced in the field of software modelling as the *Software Model Management* (SMM) problem.

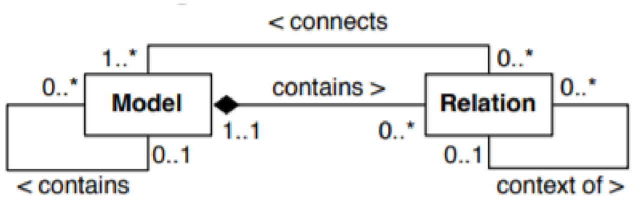


Fig. 2.1. Core meta-model of mega-models [**22**]

SMM mainly addresses the complexity caused by the interconnection of several models. It provides a high level overview to manipulate models and their relationships using transformations and operators [**8**]. SMM approaches use a special kind of models, *Mega-models*, to represent sets of models and the relationships between them [**34**]. In fact, a *mega-model* is a model whose elements are models and relationships between them [**22**]. The core meta-model of mega-models is presented in Fig 2.1.

2.2. Related work on uncertainty modelling

The Uncertainty Principle in Software Engineering [**47**] states that *Uncertainty is inherent and inevitable in software development processes and products.* However, uncertainty is rarely captured in models. Therefore, several research works have tried to explicitly articulate and characterize uncertainty.

MAVO [**33**] is a set of four different partiality-related annotations that can be used with arbitrary modelling languages. In [**17**], Famelis et al. proposed MAV-Vis, a notation for partial models based on MAVO. They evaluated it on UML Class Diagrams and Entity-Relationship Diagrams.

In [**41**], the authors attempt to harmonise the terminology and the typology of uncertainty in model-based decision support. They provide a conceptual framework for the systematic treatment of uncertainty, from a modeller's perspective, in order to improve its management in the decision making process. They suggest that uncertainty is a three dimensional concept defined by its nature, its level and its location. Although their work resides at a higher level of abstraction, their uncertainty matrix can be seen as a tool for typifying uncertainties in terms of the three dimensions. Thus, the matrix they propose constitutes a snapshot of the uncertainties in a system at a particular point in time.

In [**44**], Zhang et al. propose U-Model: a conceptual model for uncertainty specifically designed for Cyber-Physical Systems (CPSs). In another work [**46**], the same authors extend the Restricted Use case modelling (RUCM) methodology [**43**] and tool to identify and specify uncertainty as part of system requirements. They call it U-RUCM. They prove the efficiency of their method based on two Cyber-Physical Systems industrial case studies.

In [**47**], the authors propose a mathematically sound technique for modelling uncertainty based on Bayesian belief networks (presented in section 5.2.2). The authors argue that the Bayesian Network graph structure matches the one of the software systems: the belief values are associated with the software artifacts and the probability matrices are associated with the relations. Multiple Bayesian networks can be associated with one software system because multiple belief values can be associated to one software entity. The Bayesian approach can also cope with the dynamic updating of beliefs during software development thanks to its underlying Bayesian updating technique.

Partial models [**12**] are modelling artifacts that are capable of precisely and compactly encoding a set of alternative designs. The DeTUM model (Design Time Uncertainty Management Model) [**15**], is an uncertainty-aware methodology to manage design-time uncertainty using partial models.

We explore more in detail the aspects of the above works that we build upon below.

2.2.1. Uncertainty modelling in model-based decision support

In [**41**], the authors suggest that uncertainty is a three dimensional concept defined by its nature, its level and its location, as sketched in Fig 2.2.

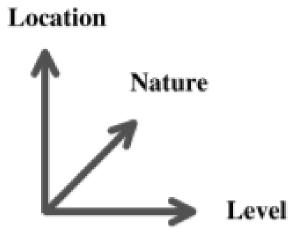


Fig. 2.2. Uncertainty as a three dimensional concept [**41**]

The authors perceive the uncertainty level as a continuous progression between determinism and ignorance, as captured in Fig 2.3. The level of uncertainty is therefore where the uncertainty manifests itself along the spectrum. The authors distinguish between the following uncertainty levels:

- **Determinism** refers to the ideal situation, where everything is known precisely.
- **Statistical Uncertainty** refers to any uncertainty that follows a statistical distribution.
- **Scenario Uncertainty** refers to the existence of a set of possible alternative scenarios. The uncertainty resides in the fact that we don't know which alternative to choose.
- **Recognized Ignorance** refers to the fundamental uncertainty, where we acknowledge that we don't know something.
- **Indeterminacy** also called **Irreducible ignorance** is at the edge of the Recognized Ignorance. It refers to the situation where it is impossible to resolve the ignorance due to its indeterminate nature.
- **Total Ignorance** means that we don't know that we don't know.

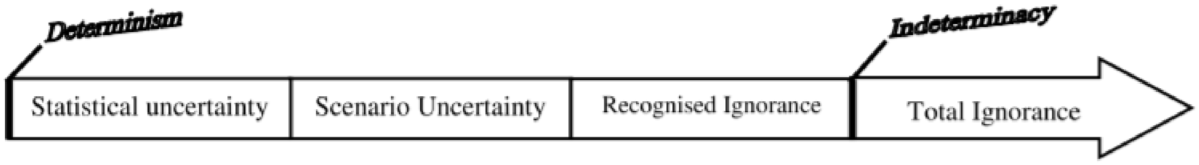


Fig. 2.3. The uncertainty levels spectrum [**41**]

2.2.2. Uncertainty modelling for Cyber-Physical Systems (CPSs)

In [**44**], Zhang et al. propose U-Model: a conceptual model for uncertainty specifically designed for Cyber-Physical Systems (CPSs). U-Model includes a *BeliefModel*, a *MeasureModel* and an *UncertaintyModel* and is mapped to the CPSs three logical levels: Application, Infrastructure, and Integration. Figures 2.4, 2.5 and 2.6 present respectively an overview of the U-Model, the *UncertaintyModel* and the *BeliefModel*.

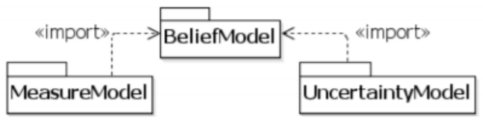


Fig. 2.4. U-Model overview [**44**]

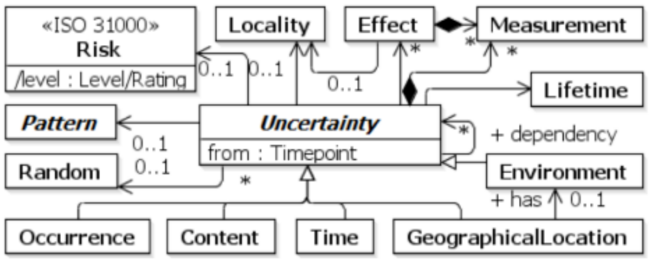


Fig. 2.5. U-Model Uncertainty model [**44**]

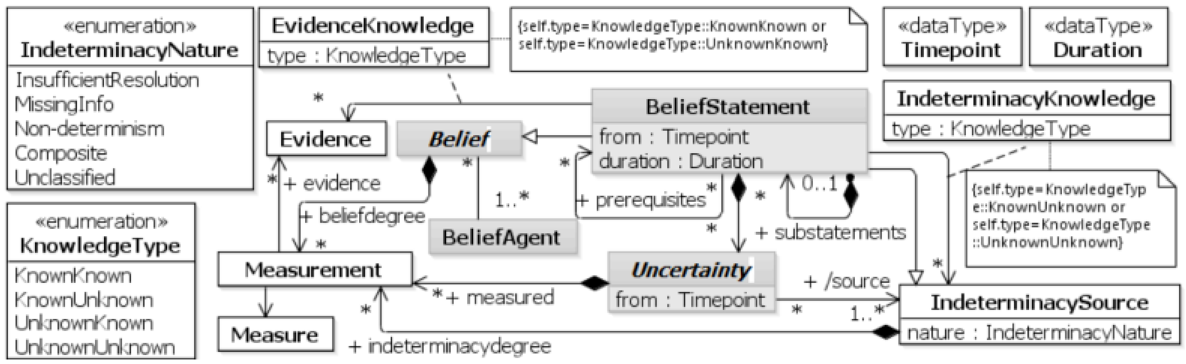


Fig. 2.6. U-Model Belief model [**44**]

The U-Model introduced several uncertainty-related concepts, among which we mention *Indeterminacy*, *IndeterminacySource* and *IndeterminacyNature*. *Indeterminacy* is an abstract concept that refers to a situation whereby the necessary knowledge is unavailable. *IndeterminacySource* is the only concretization of an *Indeterminacy*. It represents the factors that lead to an uncertainty. As there are several kinds of *IndeterminacySources*, the authors propose to categorize them as different *IndeterminacyNatures*. This categorization is presented in Table 2.1.

Indeterminacy Nature	**Definition**
Insufficient Resolution	The available information is not precise enough.
Missing Information	Some related information is unavailable.
Non-determinism	The phenomenon is non-deterministic.
Composite	A combination of two or more indeterminacy natures.
Unclassified	None of the above.

Table 2.1. Indeterminacy Natures categorization [**44**]

For example, a modeller is working on the design of some system. The modeller has already received the specifications from the client. In his design, the modeller wants to use multi-inheritance. However, he is aware that some object oriented programming languages do not support this feature. Thus, he is uncertain about this design-decision: *Should he use multi-inheritance?*. Besides, the specification file he received does not mention any implementation details. In this situation, the *IndeterminacySource* refers to the fact that the specifications file does not contain any implementation details, and its *nature* is of type *Missing Information*.

2.3. Partial models

In this section, we define partial models as well as their related *MAVO* and DeTUM frameworks.

2.3.1. Partial models as modelling artifacts

Partial models were first introduced in [**14**] as novel development artifacts capable of precisely and compactly encoding a set of alternative designs. Partial models were meant to capture design-time uncertainty, and allow developers to work in its presence without having to make a premature decision. We note that design-time uncertainty in the context of partial models refers to a set of alternative design solutions. Thus, partial models help modellers work with all the possible solutions and differ the design decision making to a later time. Each alternative solution is called a *concretization* of a partial model. The process of reducing uncertainty in partial models is called *refinement*. The end point of a refinement process is the obtention of a concrete model.

Fig 2.7 presents an example of a partial model concerning a network controller. Fig (a) represents the part of a system that the modellers are certain about. Fig (b)-(d) represent the alternative potential designs. Fig (e) is the partial model. The dashed-elements in the model are optional elements. They are *Maybe-Annotated* elements, as they may or may

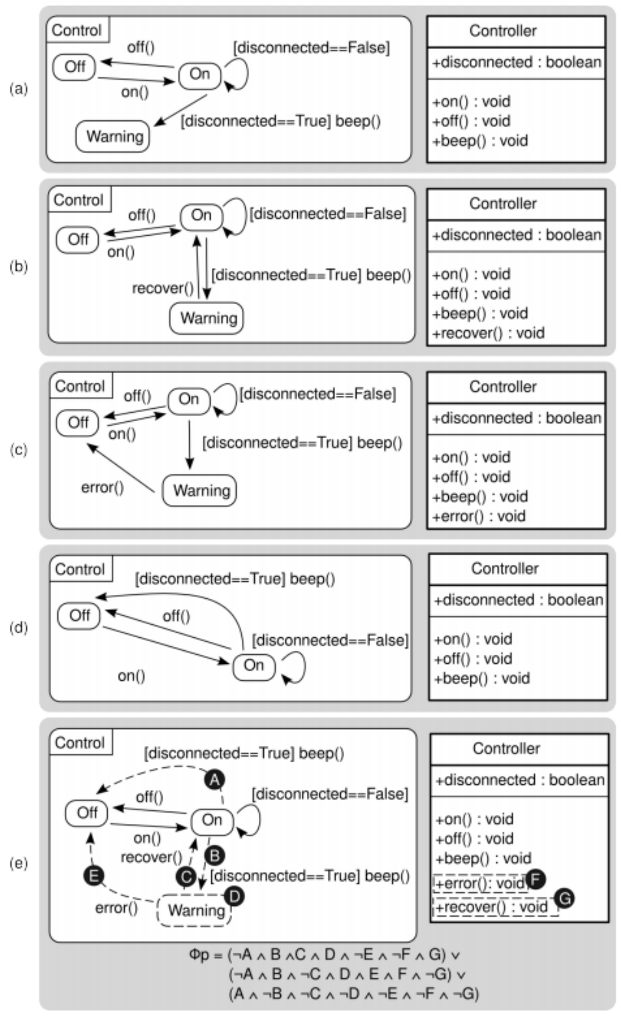


Fig. 2.7. A partial model (e) of a network controller and its set of concretizations (b)-(d) [**14**]

not exist in the final design choice. We also note the Boolean formula associated with the model. That formula is called the *May* formula and is used to capture the set of allowable configurations of the optional elements.

2.3.2. *MAVO* Framework

As mentioned above, partial models only deal with a special kind of partiality, which is the *May* partiality as defined in the *MAVO* framework [**33**]. However, there exists other types of partiality. In fact, the *MAVO* framework distinguishes four partiality types:

(1) ***May* partiality**: *May* partiality allows modellers to express how certain they are about the presence or not of particular elements in the model, by annotating them as "may" elements.

(2) ***Abs* partiality**: *Abs* partiality allows modellers to express their uncertainty about the uniqueness of the elements in the model, by annotating them as a "set" of elements or a "particular" element.
(3) ***Var* partiality**: *Var* partiality allows modellers to express their uncertainty about the distinctness of individual elements in the model, by annotating them as "constants" or "variables".
(4) ***OW* partiality**: *OW* partiality is a model-level partiality that allows modellers to express the completeness or not of the model, bu annotating it as "COMP" or "INC".

2.3.3. DeTUM **Framework**

The DeTUM model (Design Time Uncertainty Management Model), is an uncertainty-aware methodology to manage design-time uncertainty using partial models [**15**]. The DeTUM model consists of three stages: the *Articulation* stage, during which a set of candidate solutions is elicited; the *Deferral* stage, during which developers use partial models as modelling artifacts to avoid making premature decisions, and the *Resolution* stage; during which the modellers incorporate new information in the partial model in a systematic way. Fig 2.8 presents the level of uncertainty in the different stages of the DeTUM model.

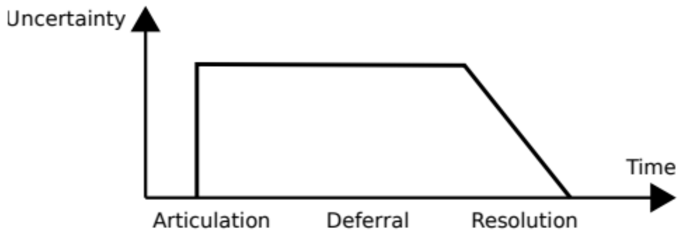


Fig. 2.8. The evolution of uncertainty throughout the DeTUM stages [**15**]

The DeTUM model is far from being a firm sequence of the above stages. In the contrary, DeTUM stages may overlap or happen in different orders. Fig 2.9 summarizes all possible transitions between the stages.

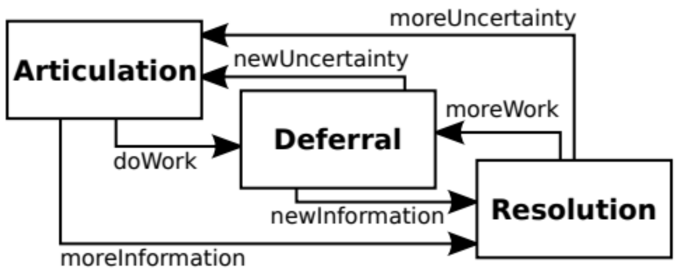


Fig. 2.9. The transitions between the DeTUM stages [**15**]

2.4. Decision-oriented modelling

Approaches for modelling decisions include the OMG's Decision Model and Notation (DMN) [**27**] as well as several product line oriented decision modelling languages.

2.4.1. The OMG Decision Model and Notation (DMN) standard

The OMG's *Decision Model and Notation (DMN)* [**27**] is a standard issued in the aim of providing the needed constructs to model decisions. Before its introduction, decision-making was uniquely addressed from two perspectives based on the existing modelling standards: business process models and decision logic models. DMN mainly introduced a third perspective; the *Decision Requirements Graph (DRG)* and its corresponding notation; the *Decision Requirements Diagram (DRD)* that bridges between the other existing two perspectives. DRDs are intended to define the decisions to be made in the tasks of the business process models, their interrelationships, and their requirements for decision logic [**27**]. However, the standard stresses out that DMN is not dependent on business process models and can be used separately.

A DRD is composed of *elements* that constitute the domain of decision-making and the dependencies between them. The dependencies express three types of *requirements*: the *Knowledge Requirement*, the *Authority Requirement* and the *Information Requirement.* The *Information Requirement* dependency refers to the idea that an output of a decision is used as input to another decision.

2.4.2. Decision-oriented modelling in Product Lines

In this section, we present decision modelling in the context of Software Product Lines Engineering. We start by introducing the *Decision-Oriented Product Line Engineering for effective Reuse* approach, then we focus on the common parts of different modelling approaches.

DOPLER (Decision-Oriented Product Line Engineering for effective Reuse [**5**]) is a variability modelling approach for Software Product Line Engineering (SPLE). SPLE is a set of methods, tools and techniques for modelling and managing families of similar software products that have slight variations [**29**]. DOPLER was proposed to compensate the rigidity of the existing variability modelling approaches, as it easily allowed domain-specific adaptations.

DoplerVML is a modelling language for product lines definition [**6**] based on DOPLER. DoplerVML's meta-model is presented in Fig 2.10. The language distinguishes between two key concepts: the *Decisions* and the *Assets.* The decisions are used to represent the problem space, e.g the available customization options, while the assets are used to define the solution space, e.g the parts required to compose the product. In other words, the decisions are

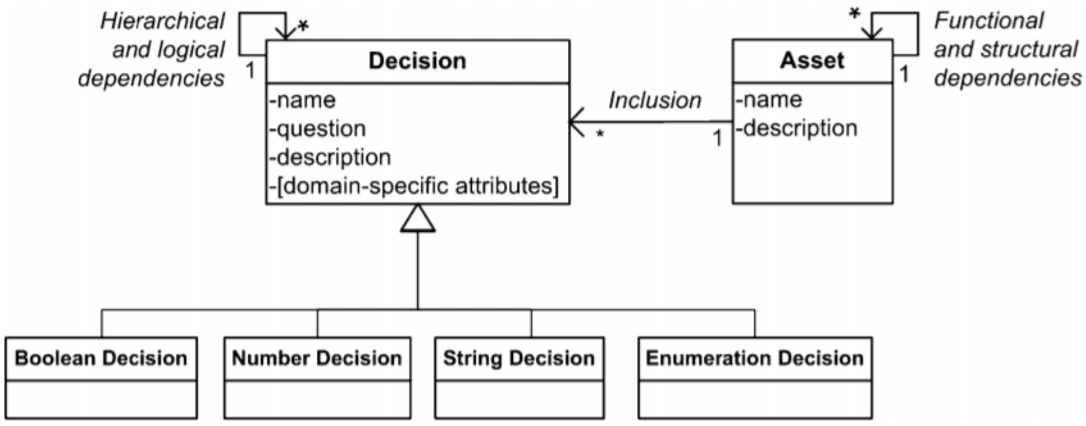


Fig. 2.10. DoplerVML meta-model [**6**]

defined for the variable parts of the product line. The model also defines traceability links between the two concepts. Thus, the customer-specific product configuration is generated automatically based on the user's input in the decisions values.

In [**35**], the authors review several decision modelling approaches for product lines, including DOPLER, and define their common elements as a basic model structure. This structure is presented in Fig 2.11.

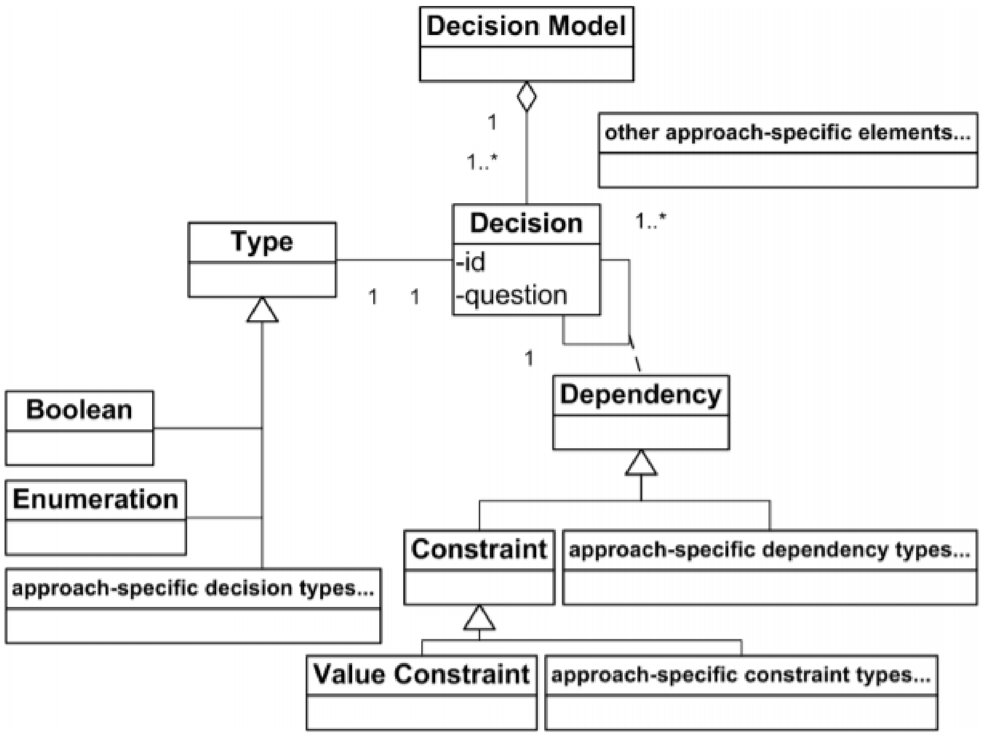

Decision Model
1
1..*
other approach-specific elements...
Type
Decision
-id
-question
1..*
1
1
1
Dependency
Boolean
Enumeration
approach-specific decision types...
Constraint
approach-specific dependency types...
Value Constraint
approach-specific constraint types...

Fig. 2.11. Common meta-model elements for decision modelling in product lines [**35**]

From Fig 2.11, we can note that all approaches define a *Decision* element with a *question* attribute that describes the decision to the user. All approaches also agree that there exists different *types* for decisions, and all of them support the *Boolean* and *Enumeration* types. Finally, all approaches allow creating *dependencies* between decisions, and agree that there can be a hierarchy of dependencies types.

In this chapter, we introduced several existing concepts from the modelling literature. In the next chapter, we detail how we used these concepts in our work.

Chapter 3

Introducing the Druide Language and Methodology

In this chapter, we start by describing the modellers needs that our work addresses. Then, we introduce our proposal that we call Druide; Design and Requirements Uncertainty Integrated Development Environment. First, we present Druide modelling language and we show how it integrates different concepts from previous works in a coherent way. Afterwards, we detail our methodology to use it and we utilize the PTNs running example (section 1.4) for illustration.

3.1. Specifications

In order to propose a language and a methodology to articulate design-time uncertainty, we focus on the modellers needs.

First, modellers should be able to express that they are uncertain about how to design some aspects of the model. This could be as vague as necessary, even if they have not figured out how their uncertainty would impact the model. Then, as the modellers understand better the implications of their uncertainty, they should be able to elaborate their original statements into more concrete design decisions. We anticipate that decisions can evolve and depend on each other.

In addition, modellers should be able to express how their uncertainty and decisions are related to the model under construction. Specifically, modellers should be able to say where their uncertainty is located in the model. This could be specific model elements, or the entire model. Moreover, if they have elicited more specific design decisions, they should be able to express how these decisions can be made operational in the model.

In the rest of this chapter, we present Druide and show how it responds to these specifications.

3.2. Language Definition

In this section, we introduce the Druide language. Fig 3.1 presents the Druide meta-model.

We start by noticing that Druide model distinguishes between two main concepts: *DUncertainties* and *DDecisions*. *DUncertainties* are objects representing the uncertainty of a modeller about the design of a software artifact, while *DDecisions* represent decisions to be made about the design. Each *DDecision* is related to one *DUncertainty*. This separation is inspired from the distinction between the problem space and the solution space in DoplerVML as presented in section 2.4.2

The *DUncertainty* specializes the *Uncertainty* element of the U-Model presented in section 2.2.2 by adding a *description* attribute to provide the modeller with the ability to express his uncertainty. Similarly, the *DIndeterminacySource* specializes U-Model's *IndeterminacySource* element by adding a *description* attribute so the modeller can explain what caused the uncertainty. Besides, the *DIndeterminacySource* has a *nature* attribute of type *DIndeterminacyNature*. *DIndeterminacyNature* is an enumeration that adds *Untrustworthiness* to the other indeterminacy natures presented in Table 2.1. *Untrustworthiness* means that although the information exists, we are not certain that it can be trusted. *Untrustworthiness* can be considered as an *Insufficiant Resolution* when the information about the source of the information is not precise enough. However, we opted to separate it as another nature in order to make the distinction between the two clear: *Insufficiant Resolution* happens when the uncertainty concerns the information, while *Untrustworthiness* is the case when the uncertainty is about the source of the information.

Moreover, the *DIndeterminacySource* specializes U-Model's *IndeterminacySource* element by adding a *level* attribute of type *DUncertaintyLevel*. *DUncertaintyLevel* is an enumeration that captures the uncertainty levels proposed by Walker et al. [**41**] as detailed in section 2.2.1. The *DUncertaintyLevel* enumeration neither includes the *Determinism* level because it refers to the absence of uncertainty, nor the *Indeterminacy* level because the modelling task in software engineering is deterministic by nature. In fact, although there might be some uncertainty during the modelling process, the modeller should be able to resolve it and deliver the final design by the end. Finally, the *DUncertaintyLevel* enumeration doesn't include the *Total Ignorance* level because we simply can't model the uncertainty that we are unaware of.

The *DDecision* element is inspired from the DoplerVML language presented in section 2.4.2. A *DDecision* has a *question* attribute that textually represents the question that must be answered for a *DDecision* to be considered resolved. Besides, it has a *resolved* Boolean attribute that indicates whether the decision has been made. A *DDecision* also has an *allowedPartiality* attribute of type *DPartiality*. *DPartiality* is an enumeration that captures

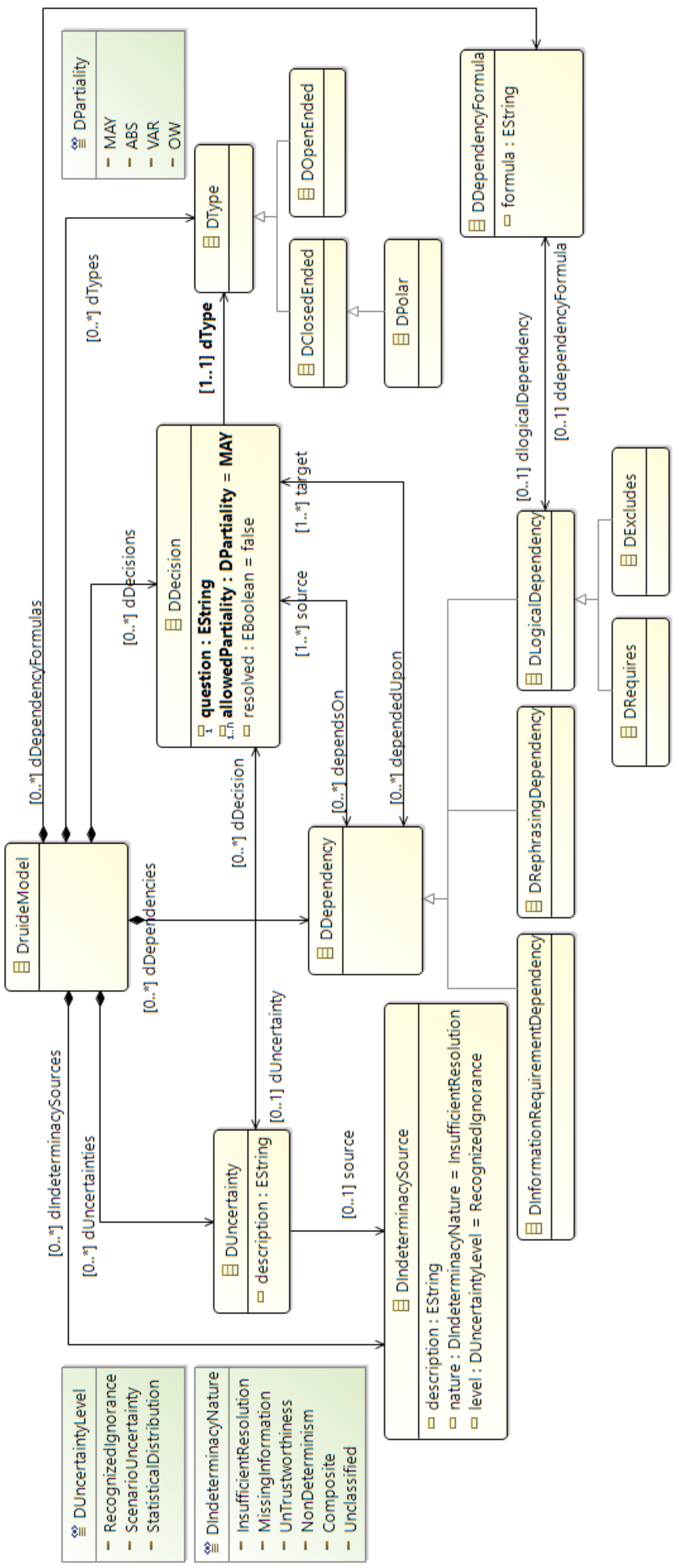


Fig. 3.1. Druide meta-model

MAVO different partiality types as introduced in section 2.3.2. For the scope of this thesis, we only consider the *MAY* partiality type.

The *DDecision* concept in Druide is inspired from the variability modelling languages. It defines similar concepts to those commonly found among decisions-oriented meta-models, namely the *types* and *dependencies* concepts, as we detailed in section 2.4.2. In fact, a *DDecision* can be of different types. Druide introduces a hierarchy of possible *DTypes* that include a *DPolar* type that means a Boolean decision, a *DClosedEnded* type that means an Enumeration decision (The decision is answered by selecting an answer from a predefined set of possible alternative answers) and a *DOpenEnded* type that means resolving the decision is still vague (There is no set of alternative answers).

Besides, Druide introduces a hierarchy of *DDependencies* between *DDecisions*. A *DDependency* can link several *DDecisions* and can be whether a *DLogicalDependency*, a *DRephrasingDependency* or a *DInformationRequirementDependency*. A *DLogicalDependency* is used when there is a logical dependency between a *source* set of decisions and a *target* set of decisions. The *DLogicalDependency* has an associated *DDependencyFormula* that is used to express the propositional logical formula that defines the dependency. We articulate two special cases for the *DLogicalDependency*: *DRequires* and *DExcludes*. *DRequires* means the *source* set of decisions requires the *target* set of decisions to be made, and *DExcludes* means the *source* set of decisions excludes the *target* set of decisions from being made.

The *DRephrasingDependency* is used when a *DDecision* rephrases another. This rephrasing can be a simple *reframing* that describes the decision from another perspective and conserves the same *DType*, or it can be a *refinement*. In the latter case, there is an evolution from the source *DDecisions* types to the target *DDecisions* types. This *DType* evolution can be from a *DOpenEnded* to a *DClosedEnded*, or from a *DClosedEnded* to a *DPolar*.

The *DInformationRequirementDependency* is inspired from the DMN standard (section 2.4.1). This kind of dependency is to be used in order to express the situation where a set of decisions output (the target set) is used as input for another set of decisions (source set).

In order to assure the well-formedness and consistency of the resulting Druide model, we introduce a set of dependency-focused constraints presented in Table 3.1.

Constraint	**Rationale**
The same set of decisions cannot depend on itself.	Otherwise, it will not make sense.
There is no cycle dependencies of the same type.	Avoid cycle dependencies.

The same sets of decisions can't have more than one dependency between them	Druide introduces a hierarchy of dependencies that is intended to capture all possible dependency types between a set of decisions.
A set of only *DPolar* decisions can't be rephrased.	*DPolar* decisions (binary questions) should be answered by yes or no. They are the final result of the thinking process. The decision should be clear enough at this point.
Logical dependencies can only exist between sets of *DPolar* Decisions.	Only *DPolar* decisions can be expressed as logical propositions. (We will detail our work in progress concerning Druide semantics later in Chapter 8.)
A *DRephrasing* Dependency only links *DDecisions* (the source and target sets) with the same *DUncertainty*.	When a set of decisions rephrases another set of decisions, they should all concern the same uncertainty.

Table 3.1. Druide dependency constraints

Since providing modellers with only a language is not sufficient, in the next section, we introduce our methodology for articulating design-time uncertainty using Druide and for working with models that contain it subsequently.

3.3. Methodology Definition

In this section, we introduce our operator-based methodology for modelling in the presence of uncertainty. We start by giving an overview, then we introduce our atomic articulation operators in detail. Afterwards, we propose a workflow and illustrate its application on the PTNs example.

3.3.1. Overview

The key idea of our approach is to work simultaneously with heterogeneous models. Thus, it is based on model management and mega-modelling (section 2.1). Mainly, it consists of using the Druide modelling language simultaneously with any other modelling language, and linking them using traces.

To do so, we define a mega-model that encompasses both models meta-models and defines the relationships between them. This resulting meta-model is sketched in Fig 3.2.

The resulting mega-model is composed of three parts: the domain modelling language is used to model some system, Druide is used to express the design-time uncertainties and their corresponding decisions, and the traces are used to localize the uncertainties in the system and to elicit the decisions in terms of the system elements. This distinction between

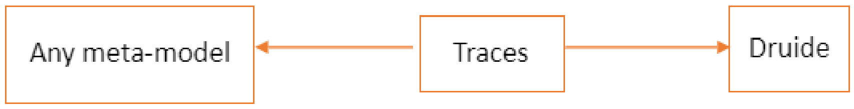


Fig. 3.2. Design-time uncertainty aware language mega-model

both spaces (the uncertainty space and the system space) is similar to the idea of separation between the problem space and the solution space presented in DoplerVML (section 2.4.2).

The traces are simply objects that reference two other objects. We distinguish between *Operationalization Traces* and *Localization Traces*. We show an illustrative meta-model fragment for the traces in Figure 3.3.

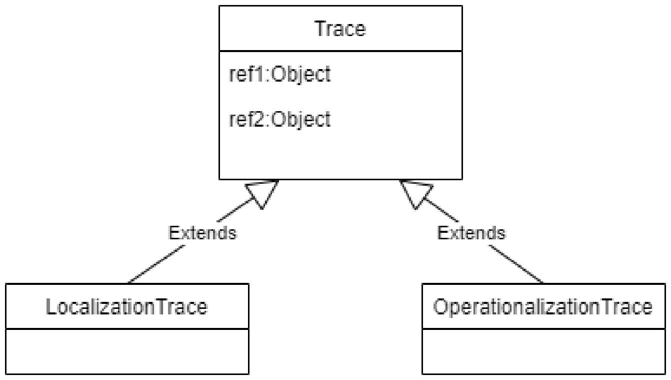


Fig. 3.3. Traces meta-model fragment

The action of creating a *Localization Trace* is called *Localization* while we refer to the action of creating an *Operationalization Trace* by *Operationalization*. In order to provide a sound basis for our methodology, we define *Localization* and *Operationalization* as atomic operators for the articulation of uncertainty in the next section.

3.3.2. Articulation operators

Our methodology to articulate uncertainty is built upon two atomic operators. We define them below:

- *Localization*: *Localization* of an uncertainty simply means identifying the parts of the system that the uncertainty concerns. *Localization* is done using Localization Traces.
- *Operationalization*: We borrow the concept of *Operationalization* from the requirements engineering field, where the leaf-level goals of a goal model are operationalized into tasks [**39**]. In a similar mindset, we only allow *Operationalization* for the last evolution level of the decisions, e.g, for the *DPolar* decisions. We mean by the *Operationalization* of a *DPolar* design decision its reflexion in the existence of a set of system elements. Thus, Operationalization Traces are used to link a *DPolar* decision

with the corresponding elements that elicit it, e.g, represent it in terms of the system elements. As mentioned above, in the scope of this thesis we are only considering the *MAY* partiality type. The action of operationalizing a *DPolar* decision by a set of elements also corresponds to annotating them as *Maybe* elements (section 2.3). We will explain the reason for this later in Chapters 7 and 8.

In order to regulate the use of the operators and check their correct usage, we define a set of constraints listed in Table 3.2.

Operator	**Constraint**	**Rationale**
Localize/ create a localization trace object	All uncertainties should be localized (e.g should at least have one localization trace).	Avoid having extremely vague uncertainties. The user should at least be able to denote which parts of the system the uncertainty relates to. In the worst case, the user can localize the uncertainty at the model level.
	An uncertainty can't have two localization traces with the same target element, e.g same source and same target.	Avoid redundant links
Operationalize/ create an operationalization trace object	Only *DPolar* decisions can be operationalized.	Other types of decisions can't be operationalized, they are still at a higher thought level.
	A *DPolar* decision can't have two operationalization traces with the same target element, e.g same source and target.	Avoid redundant links
	Every operationalization trace target should be linked to at least one of the uncertainty localization traces' target elements	The elements that operationalize a decision should be traced to the elements where that decision's uncertainty was localized. This is to make sure, to some extent, that the operationalization and the localization are consistent.

	The source *DPolar* decisions of a *DInformationRequirementDependency* can't be operationalized if one of its targets is still unresolved	A *DInformationRequirementDependency* means that the output of the target decisions set is used as input for the source decisions set. Therefore, the source set can't possibly be operationalized unless the target set has been already resolved.

Table 3.2. Constraints regarding the atomic operators

In order to illustrate better the usage of the above operators, we present an exemplar modelling workflow in the next section.

3.3.3. Modelling workflow

In this section, we propose a workflow to model in the presence of uncertainty and illustrate it using the PTNs example.

The workflow we propose is as follows:

(1) Executing a modelling task.

(2) When faced with an uncertainty, model it as a *DUncertainty* and optionaly characterize its *DIndeterminacySource*.

(3) Localize the *DUncertainty* element modeled using a localization trace. A design-time uncertainty can be localized at different parts of the system e.g, it can have multiple localization traces.

(4) Check the *Localization* constraints.

(5) Describe the *DDesicions* that you need to make and characterize them.

(6) Use the *DRephrasingDependency* associations to evolve your decisions.

(7) Use the *DInformationRequirementDependency* when the input of some *DDesicions* depends on the output of others.

(8) Use the *DLogicalDependency* associations to model any logical dependency between your *DPolar* decisions.

(9) Check the dependency constraints.

(10) If you have a clear idea about how to implement a *DPolarDecision*, use the operationalization traces to operationalize it by linking it to the corresponding system elements, and introduce new elements if necessary. A decision can be operationalized by several elements, e.g it can have several operationalization traces.

(11) Check the *Operationalization* constraints.

In the rest of this section, we illustrate the application of the above process on the PTNs example presented in section 1.4. Fig 3.4 presents the final Petri Net model with the modeller uncertainties and decisions.

The modeller was executing a modelling task when she faced some design uncertainty. Specifically, there are several ways to model the PTNs formalism, and the modeller is not sure which way to adopt [**15**]. So, she adds a *DUncertainty* element (*DU1*) and localizes it at the *Net* class level (blue link). She characterizes it by adding a *DIndeterminacySource* object (*DI1*) and specifying its level attribute to the *ScenarioUncertainty* value, as she faces a set of possibilities. Since the example in [**15**] does not specify the factors leading to this uncertainty, we suppose that the *DIndeterminacySource* is of type *MissingInformation*. The modeller didn't break any *Localization* constraints.

Afterwards, the modeller thinks more concretely about the design-decisions that she needs to make to finish modelling the PTNs meta-model. She comes to express the three following design-decisions:

(1) **D1:** How should Arcs be represented?
(2) **D2:** If Arcs are represented using separate meta-classes, should the arc meta-classes contain the weight attributes?
(3) **D3:** Should the meta-model enable the storage of the location of the graphical elements on the diagram?

The modeller adds her *DDecisions* to the model (*DD1*, *DD2* and *DD3*). As we can easily notice from the formulation of the design decisions above, *D2* and *D3* are *DPolar* decisions, and so the modeller models them as such.

D1 is a *DClosedEnded* decision with two *DPolar* alternatives: *Arcs are represented as separate meta-classes?* or *Arcs are represented as associations?* [**15**]. Thus, the modeller continues modelling by evolving the first design-decision (*DD1*) using a *DRephrasingDependency* (*DR1*). She represents the two alternatives *DDecisions*: *Are arcs represented as associations?* (*DD5*) and *Are arcs represented as separate meta-classes?* (*DD4*). Since these are two *DPolar* decisions, the modeller can use logical dependencies between them. She uses the *DExcludes* (*DE1*) dependency to indicate that the decisions are mutually exclusive.

We can also easily remark a clear *DRequires* dependency link between *D2* decision and the first *DPolar* alternative of *D1*: (*Are arcs represented as separate meta-classes?*). The modeller uses the *DRequires* association (*DRe1*) to represent this. The resulting model does not break any dependency constraints.

Finally, the modeller starts operationalizing the *DPolar* decisions. She operationalizes *DD4* and *DD5* by introducing new elements such as the classes *PlaceToTransitionArc* and *TransitionToPlaceArc* and the associations *src* and *dest*. Then, she proceeds to the operationalization of *DD2* by adding the *weight* attributes in the *PlaceToTransitionArc* and *TransitionToPlaceArc* classes. Finally, the modeller operationalizes *DD3* by the addition of

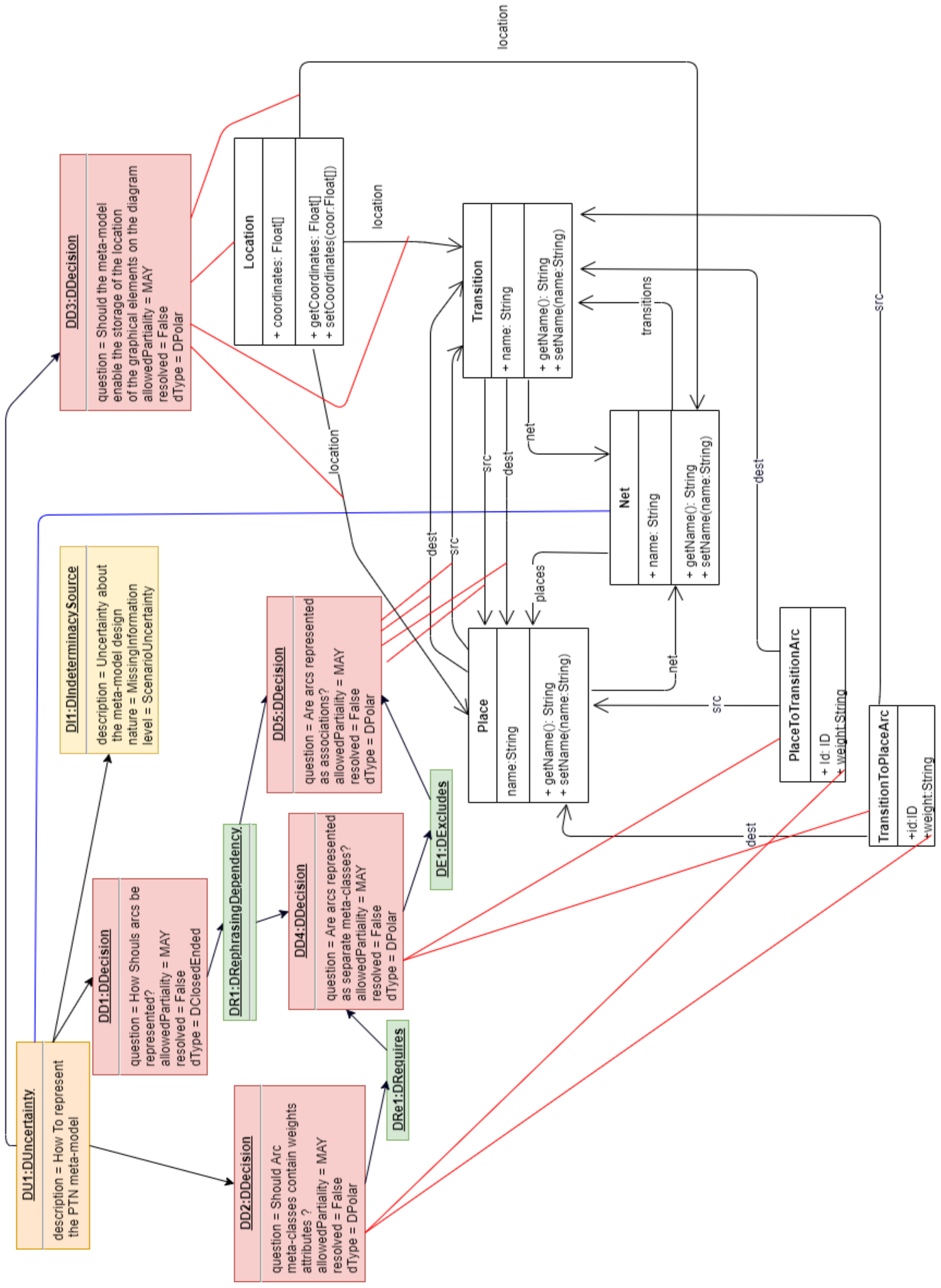


Fig. 3.4. DRUIDE applied to the Petri Nets model

the *Location* class and three *location* associations. The operationalization traces are shown in red in Fig 3.4. The modeller didn't break any *Operationalization* constraints.

As mentioned above, the *Operationalization* results in annotating the newly introduced elements as *Maybe* elements. In our example, all the elements introduced as a result of the *Operationalization* operation are *Maybe* elements. For instance, the classes *PlaceToTransitionArc* and *TransitionToPlaceArc* and the associations *src* and *dest* are *Maybe* elements. They may or may not exist in the final model.

Using the PTNs example, we have shown that DRUIDE satisfies the specifications introduced in section 3.1. In fact, the modeller was able to express that she was uncertain about how to design some aspects of the model at different levels of abstractions. She started by modelling a vague *DUncertainty* that she elaborated afterwards into a set of concrete *DPolar* decisions. The modeller was also able to evolve her decisions (for example, she evolved *DD1* to *DD4* and *DD5*), and express the dependencies between them. Finally, she was able to express where her uncertainty is located and how her specific decisions can be made operational in the model thanks to the localization and operationalization traces.

In this section, we have presented and illustrated the DRUIDE methodology. We have not yet explained how we specified the above-mentioned constraints (Tables 3.1 and 3.2) over the model. We discuss this in the next section.

3.4. Formalization of the constraints

A part of our work is to formalize the constraints in order to be able to check and constrain them over the model. In this section, we only present an illustration of the formalization of the constraints. We will present the formalization of all the constraints later in Chapter 6.

For the constraints formalization, we have chosen to use the OCL (Object Constraint Language) formalism [**28**]. OCL is a general-purpose textual language used to describe expressions on models. We have chosen OCL because it is part of the OMG (Object Management Group) standard.

To illustrate, we consider the first dependency constraint from Table 3.1: *The same set of decisions cannot depend on itself.* When the same set of decisions depends on itself, this implies the existence of some *DDependency* object that has the same set of decisions as both source and target sets. Thus, the above constraint can be obtained if we add a constraint over the *DDependency* class stating that its source must be different from its target. This can be formalized in OCL as follows:

context DDependency **inv** DepSourceNotTarget: self.source $\neq$ self.target.

In the above PTNs example (Fig 3.4), none of the dependencies (*DR1*, *DRe1* and *DE1*) have the same set of decisions as source and target sets. Thus, the above constraint holds for the resulting model.

In this chapter, we presented our approach for the articulation of design-time uncertainty and illustrated it on a worked software engineering example taken from the literature. In the next chapter, we evaluate it on two other software engineering examples.

Chapter 4

Lab-based evaluation of Druide Model and Approach

In this chapter, we begin by specifying our evaluation setup, then we focus on Druide lab-based evaluation.

4.1. Evaluation setup

In this section, we present our evaluation setup by introducing the research questions. In order to evaluate Druide model and methodology, we defined three main research questions:

- **RQ1: Adequacy.** Is Druide a language-independent modelling language and a methodology that can adequately articulate uncertainty when modelling, given the stated criteria (section 3.1)?
- **RQ2: Expressiveness.** Is Druide expressive enough for representing both intra-model and inter-model uncertainties; for both in-lab and real-life scenarios?
  We decompose this research question into four sub-questions.
  - **RQ2-1: Expressiveness for in-lab scenarios.** Is Druide expressive enough for in-lab scenarios?
  - **RQ2-2: Expressiveness for real-life scenarios.** Is Druide suitable for real-life scenarios?
  - **RQ2-3: Expressiveness for intra-model uncertainties.** Can Druide express intra-model design uncertainty?
  - **RQ2-4: Expressiveness for inter-model uncertainties.** Can Druide express inter-model design uncertainty ?
- **RQ3: Usability.** To what extent is Druide usable?

To answer the above research questions, we conducted a lab-based evaluation over two literature-based worked examples, a real-life case study and a post-study questionnaire with the three participant researchers of the case study.

In the rest of this chapter, we focus on the lab-based evaluation. The rest of the evaluation will be presented in the next chapter.

4.2. Lab-based evaluation

In this section, we present our lab-based evaluation. We start by introducing the evaluation systems and explaining our evaluation rationale. Then, we apply DRUIDE on each of the systems.

4.2.1. Evaluation systems and rationale

In order to evaluate our language and approach, we have chosen two worked examples; the Peer-to-Peer example and the UMLet Bug example, presented in [**15**] after slightly modifying them. We have chosen these examples because they are non-trivial realistic worked scenarios that have been used to validate prior relevant research work [**15**]. Since these examples were used to evaluate *May*-based partial models, e.g models working with a set of alternative designs (section 2.3), they both deal with *ScenarioUncertainty*-level uncertainties.

In the following, we evaluate each example from a different perspective in order to evaluate different aspects of DRUIDE. Thus, each example plays a different role in this evaluation:

- The Peer-to-peer example is used to highlight the fact that DRUIDE is language-independent. This example shows DRUIDE application over a behavioural model, in contrast with its application over the structural PTNs model presented in the previous chapter as an illustration. Moreover, we use the Peer-to-peer example to illustrate more the usage of the *Localization* and *Operationalization* operators, as well as to highlight the usefulness of the constraints.
- The UMLet Bug example is used to put an emphasis on the articulation and the elaboration of uncertainty rather than to focus on the mappings with the system under construction. Besides, we use the UMLet Bug example to prove the usefulness of the constraints and to detect some of DRUIDE limitations.

4.2.2. Peer-to-Peer example

In this section, we present DRUIDE evaluation over the Peer-to-peer example. The purpose of this example is to:

(1) Show that DRUIDE is language-independent.
(2) Illustrate the use of the *Localization* and *Operationalization* operators.

(3) Show the usefulness of the constraints.

In the example, a group of engineers is engaged in the modelling of a simple peer-to-peer file sharing system, called PtPP. They use UML State Machines Diagrams to model the behavior of the system. The system has three possible states: the *Idle* state which is the initial state, the *Leeching* state that refers to the state of downloading a file and the *Seeding* state during which the peer is sharing a complete local copy of a file. Both *Seeding* and *Leeching* states can be canceled by invoking the action *cancel()* and the downloading always starts from the *Idle* state. Figure 4.1 shows the corresponding object diagram.

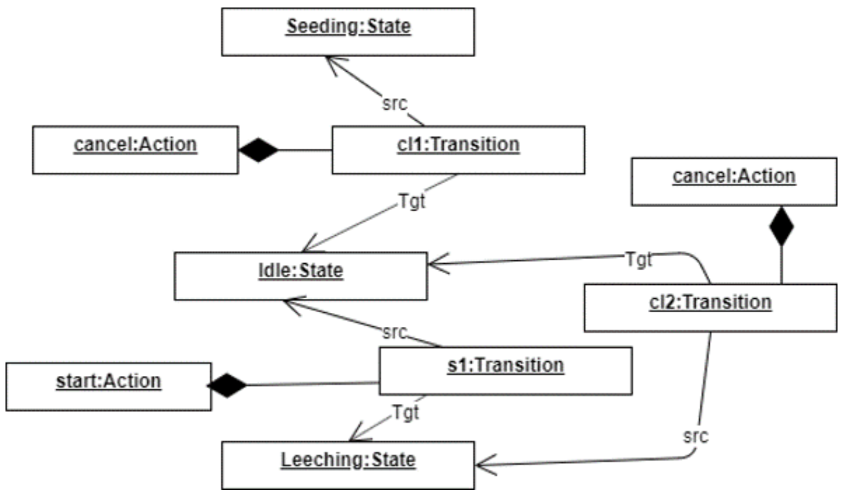


Fig. 4.1. The peer-to-peer example object diagram

The modellers have three uncertainties about other behavioural aspects of the system [**15**]. In the rest of this section, we show how the team uses DRUIDE to model their uncertainties and operationalize their decisions. The resulting object diagram is presented in Fig 4.2.

First, the team models their uncertainties as three *DUncertainty* elements:

- *DU1: How is seeding initiated?*
- *DU2: Is restarting downloads part of the wanted behaviour?*
- *DU3: How should the system behave when a download finishes?*

Since the three uncertainties are the result of the same factor, they all share the same *DIndeterminacySource* (*DI1*). The example in [**15**] does not specify the source of this uncertainty, thus we suppose it is of type *MissingInformation.* As mentioned above, we are at the level of the *ScenarioUncertainty.*

Second, the modellers use the localization traces to localize their uncertainties in the corresponding parts of the system, as represented by the blue links in Fig 4.2. For instance, *DU1* is localized at the *Seeding* state. This localization didn't break any localization constraints.

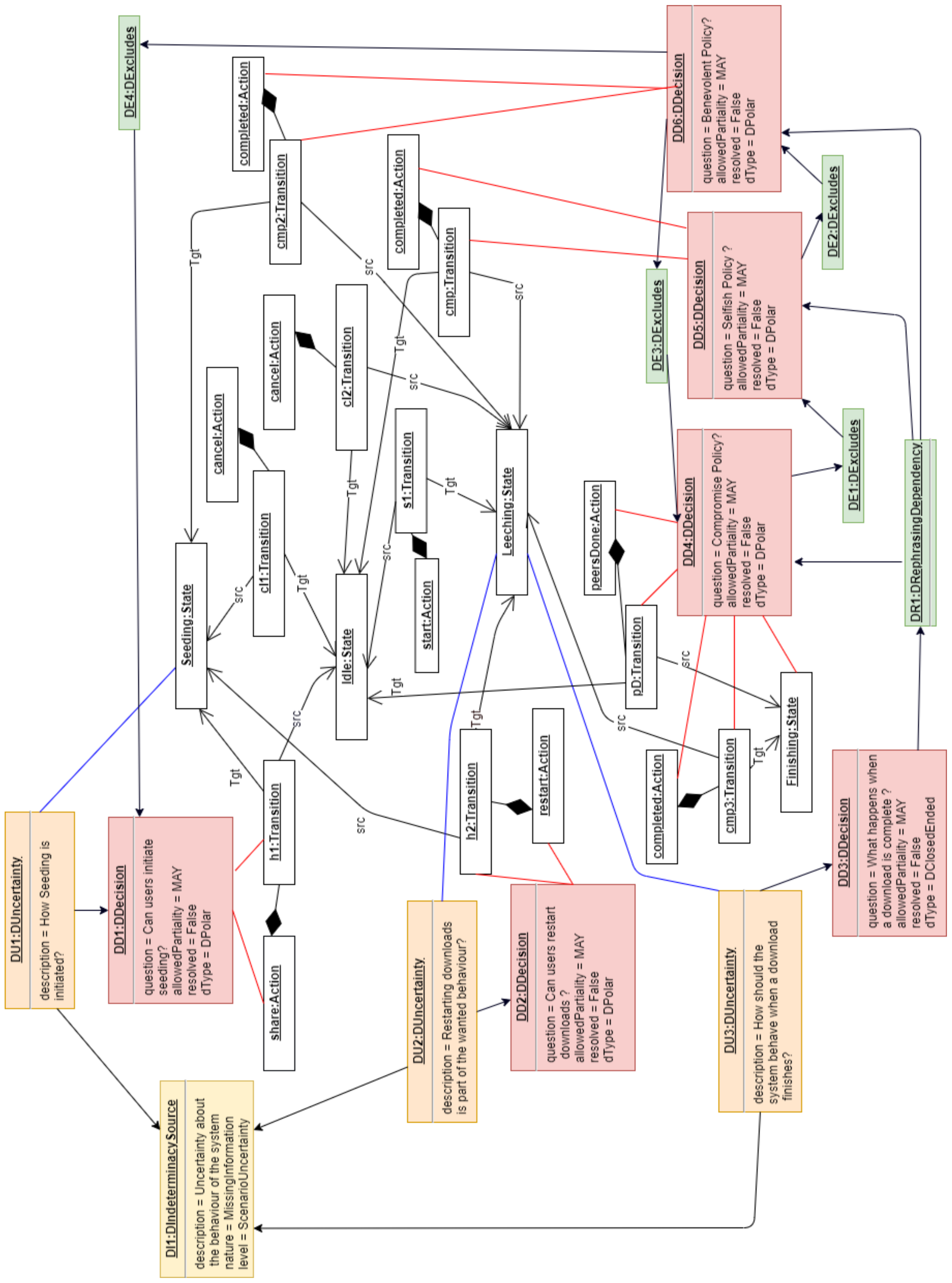


Fig. 4.2. The resulting mega-model after applying Druide on the Peer-to-peer example

Afterwards, the team articulates the specific design decisions that they need to make in order to resolve their uncertainty. Specifically, they express the following three design-time decisions that respectively correspond to the above uncertainties:

- *D1) Can users initiate seeding?*
- *D2) Can users restart downloads?*
- *D3) What happens when a download is completed?*

Besides, when it comes to the third design-decision, the team considers three potential design solutions: a *benevolent* policy in which the program automatically starts seeding once the leeching is completed, a *selfish* policy in which the program becomes idle once the leeching is completed and a *compromise* policy in which the program stops accepting new peers but doesn't disconnect from connected peers once the leeching is completed. Furthermore, the team decides to only allow the ability to start *seeding* for the *selfish* and *compromise* policies [**15**].

Now, for each uncertainty, the team models its corresponding decisions. For instance, the first design-decision *D1) Can users initiate seeding?* is a boolean decision that corresponds to the *DUncertainty DU1.* Thus, the team adds the *DPolar* decision *DD1.* Similarly, the team models the second design decision *D2) Can users restart downloads?* that corresponds to the *DUncertainty DU2* as the *DPolarDecision DD2.*

After that, the engineers model the third design-decision *D3) What happens when a download is completed?* as a *DClosedEnded DDecision DD3.* As detailed above, the team has already thought about three alternative policies to adopt when a download is complete. The modellers model this *refinement* of *DD3* using the *DRephrasingDependency DR1.* They evolve *DD3* to three *DPolar* decisions corresponding to the three policies: selfish *(DD5)*, benevolent *(DD6)* and compromise *(DD4)*.

The three policies are mutually-exclusive. The engineers express this using the logical dependencies *DE1*, *DE2* and *DE3.* Finally, as mentioned above, the team wants to allow the ability to start seeding uniquely for the *selfish* and *compromise* policies. So, they add a *DExcludes* dependency *DE4* between the benevolent policy *DDecision (DD6)* and the ability to start seeding *DDecision (DD1).*

When checking the dependency constraints, the team finds out their model (Fig 4.2) breaks the following constraint: *There is no cycle dependencies of the same type.* In fact, the *DExcludes* dependencies *DE1*, *DE2* and *DE3* form a cycle of the same type. In the final model, the modellers fix this by swapping the target and source decisions of *DE1.*

Although in this particular case, detecting a cycle of *DExcludes* dependencies may not seem very useful, the constraint of detecting cycle dependencies in general is very beneficial. For instance, semantically, detecting a cycle of *DRequires* dependencies may help detect infinite loops and recursions or prevent some unexpected failures. We will detail our work in progress concerning DRUIDE semantics later in Chapter 8.

The team now starts operationalizing the *DPolar* decisions. They operationalize *DD1* by the addition of a transition with the *share()* action from the *idle* state to the *seeding* state. Similarly, the team operationalizes the second design decision *D2) Can users restart downloads?* (*DD2*) by the addition of a transition with the *restart()* action from the *Seeding* state to the *Leeching* state. After that, they operationalize each of the three mutually exclusive policies as detailed above. The *Operationalization* traces are shown as red links in Fig 4.2. The *Operationalization* step didn't break any constraints.

In this example, we note that the modellers decided to distinguish between their uncertainties, localize each one in a different part of the system, and link each one with its corresponding *DDecisions*, in contrast with the PTNs example above, where the modeller uses one *Duncertainty* element for all her *DDecisions*, and localizes it at the *Net* class high level. Both ways are correct and supported in DRUIDE. These subjective modelling choices illustrate the flexibility of DRUIDE.

Besides, in this example and for the sake of structuring, we have presented the modelling process as a series of big steps organized as follows: modelling all the uncertainties, localizing them, checking the *Localization* constraints, expressing and evolving all the decisions, checking the dependencies constraints, operationalizing all the *DPolar* decisions and finally checking the *Operationalization* constraints. However, we note that there is no constraint for the modelling process to happen this way, e.g by applying an operation for all the same elements at once. We also admit that the process we used is not realistic. In fact, it is perfectly natural to model an uncertainty, operationlize a decision and then refine another, as the modellers are usually at different evolution stages concerning the different uncertainties. In other words, the workflow proposed in section 3.3.3 is to be applied per uncertainty.

After successfully applying DRUIDE on two literature-based examples, and showing that it satisfies the uncertainty articulation requirements (section 3.1) both on a structural model (the PTNs example) and a behavioural model (the Peer-to-peer example), we can answer **RQ1**.

RQ1 results: DRUIDE *is a language-independent modelling language and a methodology that can adequately articulate uncertainty when modelling, given the stated criteria.*

4.2.3. UMLet Bug example

In this section, we present DRUIDE evaluation over the UMLet Bug example. The purpose of this example is to :

(1) Illustrate the articulation and elaboration of uncertainty, Thus, we omit the mega-model and the traces and only present the DRUIDE uncertainty model.
(2) Highlight some of DRUIDE limitations.
(3) Show the usefulness of the constraints.

In this example, we look at the bug report *Bug #10*[1] of UMLet, an open-source Java-based UML drawing tool. In UMLet, the elements are placed on a canvas that has several layers. An element's "z-order priority" indicates what layer of the canvas it is on. As discussed in [**15**], the bug concerns the design of the copy-paste feature of UMLet. Specifically, the copy-paste feature creates a new copy but does not give it higher z-order priority.

In order to fix the bug, the modeller creates a *positioner* object that has a method *moveToTop* that places the new copy on top of others, thus giving it higher z-order priority. The fix is shown encircled by a dashed line in Fig 4.3.

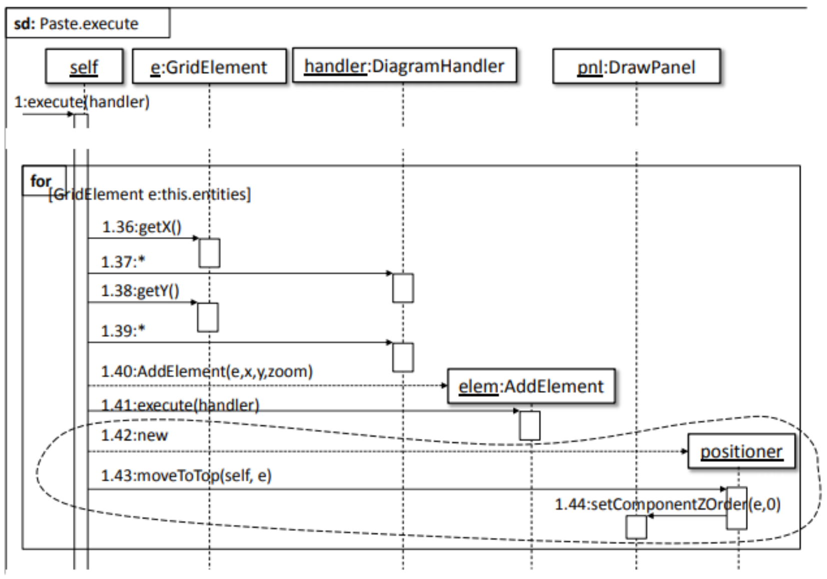


Fig. 4.3. Fragment of the sequence diagram of the UMLet paste function [**15**]

However, this fix creates additional problems as it violates two consistency rules: **ClasslessInstance** (the *positioner* object is not associated with a class) and **DanglingOperation** (the method *moveToTop* is not in the receiving object class, e.g, it is not in the *positioner* class, because it has no class.) [**15**].

So, the modeller uses an automated technique to generate alternative model repairs. The technique proposes the following repair strategies:

- Repair strategies to fix the ClasslessInstance consistency rule:

[1]Bug #10 available here: `https://github.com/umlet/umlet/issues/10`. URL accessed on September 13th, 2020.

    - Remove the object
    - Replace the object with an existing object with a class
    - Assign the object to an existing class
    - Assign the object to a new class
- Repair strategies to fix the DanglingOperation consistency rule:
    - Put the operation into the receiving object's class
    - Change the operation to another one already in the receiving object class
    - Remove the message

The modeller is uncertain about which repair to use. She tries to model her uncertainties with DRUIDE. Fig 4.4 shows the resulting DRUIDE model.

The modeller expresses her uncertainty about fixing the model using the *DUncertainty* element *DU1*. She then models the *DDecisions* she needs to make: *DD1* and *DD2*. Since the modeller has already several possible alternatives, she specifies their types as *DClosedEnded*.

The modeller refines the first *DDecision DD1* into four *DDecisions*; *DD3*, *DD4*, *DD5* and *DD6*, that correspond to the four repair strategies to fix the ClasslessInstance consistency rule specified above. Since these strategies are mutually exclusive, the modeller uses the *DExcludes* dependencies to express this.

Similarly, the modeller refines *DD2* using the *DRephrasingDependency DR2* into the three potential fixes for the DanglingOperation consistency rule listed above. She also adds several *DExcludes* dependencies to denote the fact that these solutions are mutually exclusive.

When modelling, the modeller notices that removing the object fix (*DD3*) implicitly implies removing the message fix (*DD9*). The modeller uses the *DRequires* dependency *DRe1* to express this.

The *DDecisions DD4*, *DD5* and *DD8* can't be modeled as *DPolar* because they can't be operationalized yet. In fact, they change based on some variables. For instance, in the case of *DD4*: *Should we replace the object with an existing object with a class?*, this decision depends on which object among the existing objects the modeller will choose. Thus, the modeller models it as a *DClosedEnded DDecision* as it can be rephrased to a set of decisions, with each of the possible values of the existing objects as a separate *DPolar* decision. For example, if we have N existing objects in the model each identified by a number from $1..N$, *DD4* will be rephrased into N *DPolar* decisions of this form: *Should we replace the object with the existing object Obj_i, with* $1 < i < N$.

Although the *refinement* is mainly supposed to evolve the *DDecisions* types (section 3.2), there is no strict constraint in DRUIDE about this. In this case, *DR1* evolves the *DClosedEnded DDecision DD1* into a set of *DDecisions* of the same type (*DD4* and *DD5*) and a set of *DPolar DDecisions* (*DD3* and *DD6*). This example shows DRUIDE flexibility and ability to support the articulation of uncertainty as the decisions evolve.

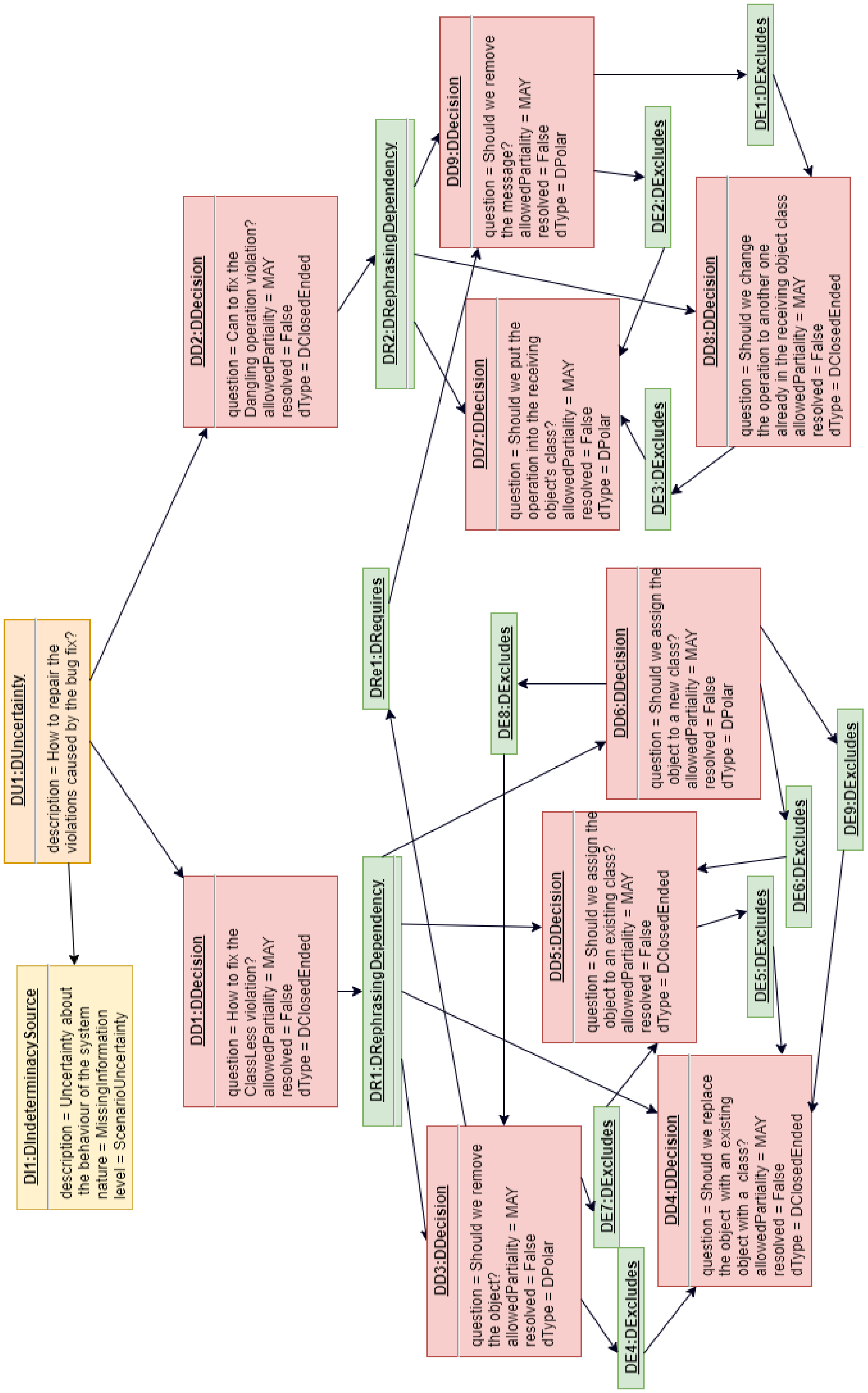


Fig. 4.4. The DRUIDE model representing the modeller's uncertainties

Besides, the concept of *DClosedEnded* decisions is broad enough in Druide that it can capture simple variable changes, like in this scenario, or significantly different design decisions like in the Peer-to-peer scenario. This is another illustration that Druide concepts are general enough to capture a variety of scenarios, and thus Druide has good support for the articulation of uncertainty at different granularity levels.

Similarly to *DD4*, the modeller specifies *DD5* and *DD8* as *DClosedEnded DDecisions* as well. *DD5* changes based on the choice of a class among the existing ones, while *DD8* changes based on two variables: the choice of the receiving object class, then the choice of the operation among the operations available in that class. So, if M is the total number of all existing operations in the model, *DD8* can be rephrased into *M DPolar DDecisions.* Besides, for each decision, there should be a set of dependencies to correlate it with the correct class choice, e.g, the choice of the operation should be consistent with the choice of the receiving object class.

Although this example is feasible with Druide, and would be very simple if the system model only has a small number of classes and operations, this is not the case of the rather large system presented in [**15**]. In fact, the Druide model will be too big and not practical to use. Besides, it will become very annoying and time-consuming for the modeller to operationalize each *DPolar* decision. We conclude that, although Druide supports expressing uncertainty at different levels of granularity, using it to express very simple model variances can be an overkill. We will discuss more Druide limitations in Chapter 7.

When thinking about her model, the modeller notices something: although *DD7* is a *DPolar* decision, it cannot be operationalized yet because it depends on the choice of the class of the receiving object. Thus, the modeller investigates the use of the *DInformation-RequirementDependency* with one source (*DD7*) and three targets (*DD4*, *DD5* and *DD6*). Mainly, she considers the constraints related to it.

The *Operationalization* operation has a constraint concerning the *DInformationRequirementDependency* (Table 3.2) that specifies: *The source DPolar decisions of a DInformationRequirementDependency can't be operationalized if one of its targets is still unresolved.* In our case, the targets *DD4*, *DD5* and *DD6* are unresolved, so the constraint makes perfect sense and conforms to the fact that the modeller couldn't operationalize *DD7* yet. This is another illustration of the usefulness of the constraints, as they ensure the well formedness and consistency of the resulting uncertainty model as well as the good usage of the operators.

After successfully applying Druide on three literature-based examples (the PTNs, the Peer-to-peer and the UMLet Bug examples), we can answer **RQ2-1**.

RQ2-1 results: Druide *is expressive enough for in-lab scenarios.*

In this chapter, we only presented the lab-based evaluation and addressed two research questions. In the next chapter, we present the rest of the evaluation and answer the remaining research questions.

Chapter 5

Field-based evaluation of Druide Model and Approach

In this chapter, we address the rest of the research questions by presenting our real-life case study and post-study questionnaire. Finally, we enumerate the threats related to this evaluation.

5.1. Case study context

In order to evaluate our proposal outside the laboratory, assess the usability of our work in a real context and receive modellers feedback and insights about Druide, we proceeded to a collaborative practical evaluation by identifying a case study from real life. In fact, our evaluation case study is related to the uncertainty produced by the COVID-19 pandemic.

For this part, we invited three researchers from the Grubb-Lab[1] at Smith College[2], USA. to collaborate with us. The Grubb-lab's main research focus is goal modelling. The three researchers have different modelling experiences; they are two students with little modelling experience less than five years, and a professor with an important experience in modelling of more than five years.

This collaboration happened over the course of three months, in the form of bi-weekly virtual meetings. The first meetings were to introduce Druide and agree on the case study. We chose one that combines two heterogeneous models, one of them is a goal model. Then, we divided the work and started modelling our uncertainties using Druide on our separate parts.

For the goal model part, the three Smith researches chose to each work alone, then gather their Druide models into one. The modelling process happened incrementally and

[1]The Grubb lab is the software engineering lab in the Department of Computer Science at Smith College. The lab research work focuses on goal modelling, and how stakeholders can effectively use models in collaboration. Website: `https://amgrubb.github.io/grubb-lab/`. URL accessed on September 13th, 2020.
[2]Website: https://www.smith.edu/. URL accessed on September 13th, 2020.

iteratively as we constantly discussed with them and gave them feedback about DRUIDE usage and concepts. This collaboration has been proven beneficial as it was the main reason the *DInformationRequirementDependency* was added to DRUIDE.

Afterwards, we merged both our models together, and discussed the potential inter-model uncertainties. At the end of this modelling experience, we asked our three participants to answer a post-study questionnaire to collect their feedback and insights about DRUIDE.

In the next sections, we start by presenting the necessary background for the case study, then we present the outcomes of this evaluation.

5.2. Case study background

In this section, we present necessary background knowledge for our case study. First, we introduce goal models, then we explain Bayesian Belief networks.

5.2.1. Goal models

Goal models are a way to capture and refine stakeholders intentions to generate functional and non-functional requirements, thus goal modelling is closely related to the requirements engineering (RE) field [**25**]. Generally, goal models are used in the early phases of projects in order to evaluate tradeoffs with stakeholders. Fig 5.1 presents the meta-model of the Tropos goal model [**3**] used in our case study.

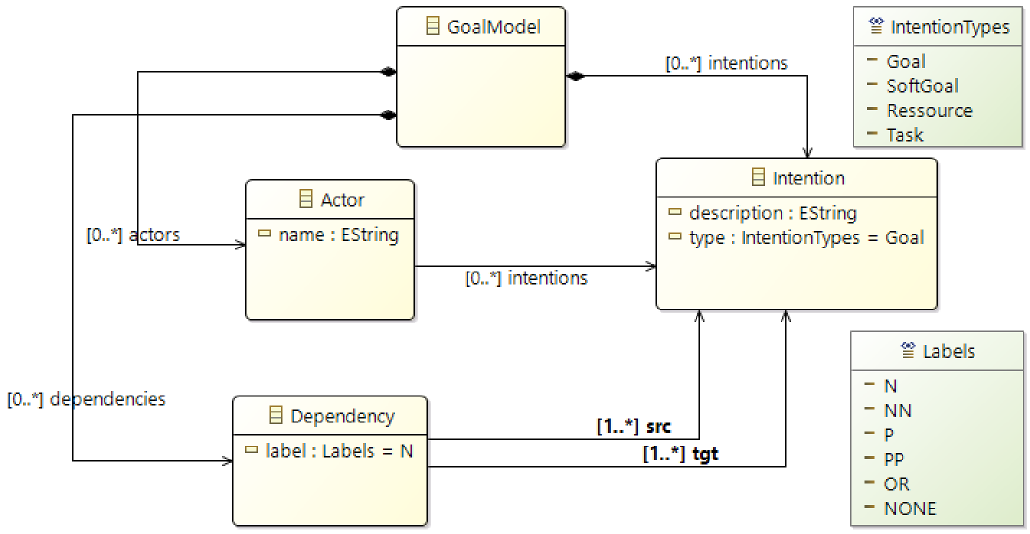


Fig. 5.1. Tropos meta-model

Tropos goal models consist of three types of elements: actors, intentions and dependencies. Actors represent stakeholders. Each actor is associated with a set of intentions. An intention can be a goal, a soft-goal, a task or a resource. The intentions are connected with labeled dependencies, also called relationships or contribution links. There are several types of contribution links such as ‘++’, ’+’, ‘–’, or ‘OR’. Each labeled relationship has a different meaning. For instance, the ’+’ contribution link means that the source intention helps the satisfaction of the target intention, whereas the ’-’ dependency means that the source intention hurts the satisfaction of the target intention.

5.2.2. Bayesian Belief Networks

Bayesian belief networks have been extensively used in artificial intelligence research for reasoning under uncertainty [**11**]. Bayesian Networks are acyclic directed graphs. The nodes represent variables on domains composing of discrete mutually exclusive values. The edges represent causal influence. Each edge is associated with a probabilities matrix that indicates

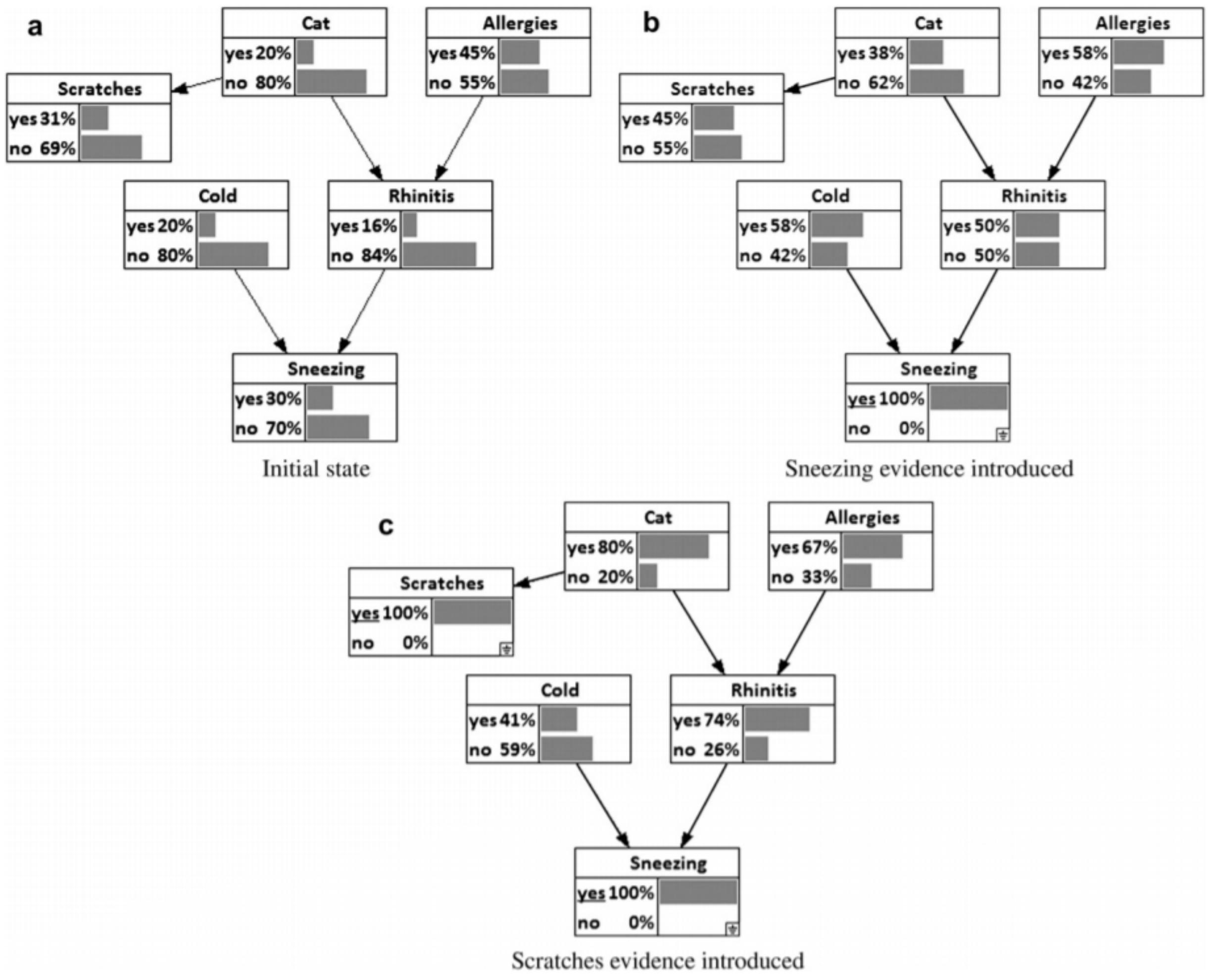


Fig. 5.2. The evolution of the bayesian network of a sneezing example as new evidence is introduced. [**26**]

beliefs in how each value of the cause variable affects the probability of each value of the effect variable. These matrices are either estimated by experts or deduced from statistical studies. The probabilities are updated each time a new evidence comes. Those probabilities can then be used to determine the most likely causes of some events *(diagnostic reasoning)* or to predict the results of some tests *(predictive reasoning)* [**26**].

Fig 5.2 shows the evolution of the parameters of a sneezing example bayesian model as new evidence is introduced [**26**]. This example represents a diagnostic model, intended to help some person find out whether he is sneezing because of a cold or due to rhinitis caused by an allergic reaction [**26**]. The example contains six Boolean variables and the causal relationships between them. Fig. 5.2-a shows the initial state, Fig. 5.2-b shows the model's updated state when the sneezing evidence has been introduced, and Fig. 5.2-c shows the model's updated state when the scratching evidence has been introduced. A live demo of another bayesian network can be found here[3].

5.3. COVID-19 Case study

In this evaluation, we focus on evaluating tradeoffs of individual decisions in the context of the COVID-19 pandemic. Specifically, we consider Emma, a persona representing a Quebec resident who wants to have dinner during the COVID-19 pandemic, without getting or transmitting the virus. Emma has some options for dinner and she wants to choose the safest option. In order to help her make the correct decision, we propose to use real COVID-19 Quebec related data that we fit to an artificial intelligence reasoning algorithm. When modelling, this case study revealed several uncertainties, which makes it is a great real example to evaluate our proposal on.

Specifically, we address this problem in three steps: first, we use goal models to model Emma's decisions and concerns (section 5.3.1). Second, we use Bayesian Belief Networks to represent the COVID-19 epidemiological model in Quebec (section 5.3.2). Third, we propose a linkage of both models that helps Emma make the safest choice (section 5.3.3). In each of the previous steps, we use Druide to articulate our uncertainties and potential modelling decisions.

The purpose of this case study is mainly to :

(1) Show the applicability of Druide in real-life scenarios.
(2) Show Druide's ability to express inter-model uncertainties.
(3) Prepare the foundation for a post-study questionnaire to collect practitioners insights about Druide.

Besides, since we have already shown the usefulness of the constraints and illustrated their usage in the previous chapter, we will not explicitly focus on them in this chapter. Instead,

[3]`https://www.bayesserver.com/examples/networks/asia`. URL accessed on September 13th, 2020.

we will be informally enforcing them throughout the case study. Finally, in the rest of this chapter, the blue links are used to denote the localization traces in the figures, while the red links represent the operationalization traces.

5.3.1. Emma's Goal Model

The work presented in this section was conducted mainly by the Smith researchers[4]. Specifically, they use a goal model to evaluate the tradeoffs decisions for Emma's goal, which is to have dinner without getting or transmitting the virus. Their model is presented in Fig 5.3.

The model has two actors: Emma and Society. Emma has two goals; *to have dinner*, and *to not get or transmit COVID-19*. Emma considers two potential tasks that can allow her to reach her first goal: *pick up takeout* or *cook at home*. The *cook at home* task has a positive relationship with the *practice social distancing* soft goal. The Society has three soft goals: *minimize economic impact*, *minimize exposure to essential workers* and *minimize the spread of COVID-19*. The second goal contributes positively to the third goal.

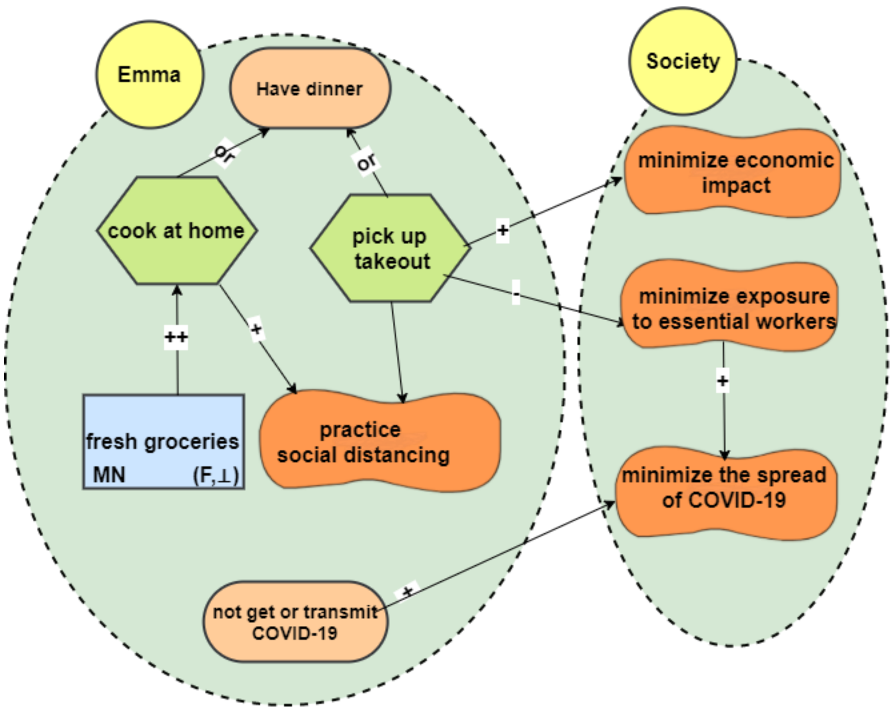

Emma
Have dinner
or
or
cook at home
pick up takeout
++
+
fresh groceries
MN (F,⊥)
practice social distancing
not get or transmit COVID-19
+
Society
minimize economic impact
+
minimize exposure to essential workers
-
+
minimize the spread of COVID-19

Fig. 5.3. Tropos goal model of Emma's decisions

Finally, there are relationships between Emma's intentions and the society's intentions. Specifically, the *pick up takeout* contributes positively to the *minimize economic impact* soft

[4]They have written a brief description of this collaboration on the Grubb Lab website. It can be found here: `https://amgrubb.github.io/posts/2020-08-20-covid-uncertainties`. URL accessed on September 13th, 2020.

goal, and negatively to the *minimize exposure to essential workers* soft goal. Besides, Emma's second goal, *to not get or transmit COVID-19* contributes positively to the *minimize the spread of COVID-19* soft goal.

The Smith researchers have several uncertainties concerning how to finish modelling the above goal model. They use Druide to describe these modelling uncertainties. The resulting model is presented in Fig 5.4.

First, the Smith modellers are not sure what is the relationship between the *pick up takeout* task and the *practice social distancing* soft goal. They model it as the *DUncertainty E-DU1.* This uncertainty resulted from the fact that they do not know to what extent does picking up takeout helps practice social distancing, since there is not a precise information that states it. Thus, they characterize the *DIndeterminacySource E-DI1* nature as *InsufficientResolution.* Since there is only a set of possible relationships in goal modelling, this is a *ScenarioUncertainty.* The modellers localize *E-DU1* using the blue *Localization* traces as shown in Fig 5.4.

To resolve this uncertainty, the modellers suggest decomposing the *pick up takeout* task into two tasks: *pick up takeout with contact* and *pick up takeout with No contact.* They model this as the *DPolar* decision *E-DD1.* However, this decomposition would result in the addition of some contribution links. The modellers thus need to decide what contributions links should be added if they decide to decompose *E-DD2.* Since they didn't make their decision about decomposing yet, and since *E-DD2* depends on the output of *E-DD1*, the modellers use the *DInformationRequirementDependency* to express this.

Second, the modellers are wondering if there exists a relationship between the *practice social distancing* soft goal and the *not get or transmit COVID-19* goal. Specifically, they are unsure if this preventive measure helps reduce the spread of the virus, has no effect at all, or in the contrary, it makes matters worse. In the absence of this information, the modellers cannot decide. Thus, they express this using the *DUncertainty E-DU2* and characterize the *DInderminacySource E-DI2* by specifying its level to *RecognizedIgnorance* and its nature to *MissingInformation.* Then, they localize it using the blue *Localization* traces.

Afterwards, the modellers articulate their decisions. Initially, they need to decide if a relationship exists (*E-DD3*). They operationalize this decision by the introduction of a link between the *practice social distancing* soft goal and the *not get or transmit COVID-19* goal. This link is connected to *E-DD3* by an *Operationalization* trace shown in red Fig 5.4. The modellers also need to decide which contribution link to use on this relationship, so they model the *DClosedEnded* decision *E-DD4.* Similarly, since the modellers didn't make their decision about the existence of the link yet, and since *E-DD4* depends on the output of *E-DD3*, the modellers use a *DInformationRequirementDependency* to express this.

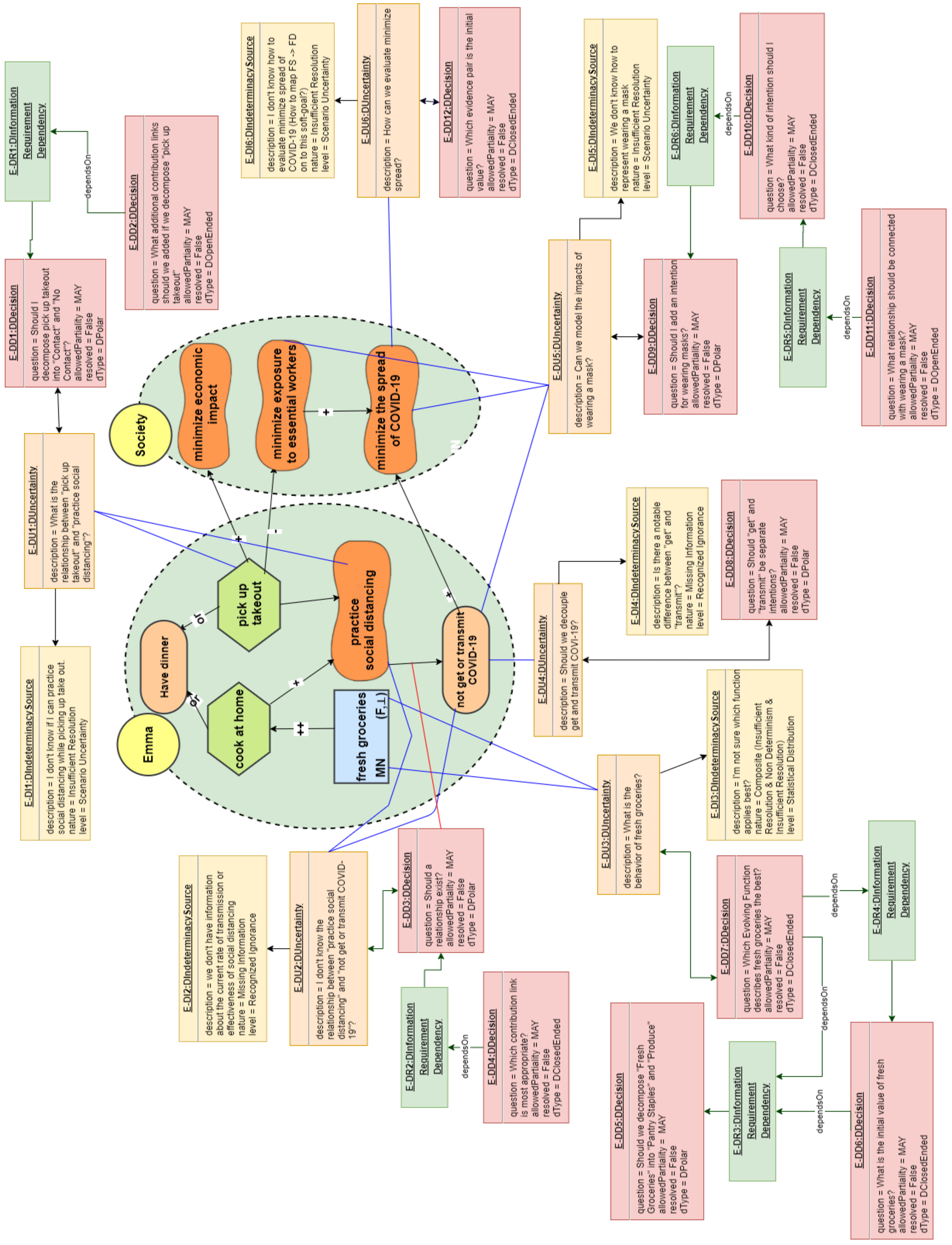

Society
minimize economic impact
minimize exposure to essential workers
minimize the spread of COVID-19
Emma
Have dinner
pick up takeout
cook at home
practice social distancing
fresh groceries MN (F,⊥)
not get or transmit COVID-19
E-DR1:DInformation Requirement Dependency
E-DD1:DDecision question = Should I decompose pick up takeout into "Contact" and "No Contact"? allowedPartiality = MAY resolved = False dType = DPolar
E-DD2:DDecision question = What additional contribution links should we added if we decompose "pick up takeout" allowedPartiality = MAY resolved = False dType = DOpenEnded
E-DU1:DUncertainty description = What is the relationship between "pick up takeout" and "practice social distancing"?
E-DI1:DIndeterminacySource description = I don't know if I can practice social distancing while picking up take out. nature = Insufficient Resolution level = Scenario Uncertainty
E-DI6:DIndeterminacySource description = I don't know how to evaluate minimize spread of COVID-19 (How to map FS -> FD on to this soft-goal?) nature = Insufficient Resolution level = Scenario Uncertainty
E-DU6:DUncertainty description = How can we evaluate minimize spread?
E-DD12:DDecision question = Which evidence pair is the initial value? allowedPartiality = MAY resolved = False dType = DClosedEnded
E-DI5:DIndeterminacySource description = We don't know how to represent wearing a mask nature = Insufficient Resolution level = Scenario Uncertainty
E-DR6:DInformation Requirement Dependency
E-DD10:DDecision question = What kind of intention should I choose? allowedPartiality = MAY resolved = False dType = DClosedEnded
E-DU5:DUncertainty description = Can we model the impacts of wearing a mask?
E-DD9:DDecision question = Should I add an intention for wearing masks? allowedPartiality = MAY resolved = False dType = DPolar
E-DR5:DInformation Requirement Dependency
E-DD11:DDecision question = What relationship should be connected with wearing a mask? allowedPartiality = MAY resolved = False dType = DOpenEnded
E-DU4:DUncertainty description = Should we decouple get and transmit COVI-19?
E-DI4:DIndeterminacySource description = Is there a notable difference between "get" and "transmit"? nature = Missing Information level = Recognized Ignorance
E-DD8:DDecision question = Should "get" and "transmit" be separate intentions? allowedPartiality = MAY resolved = False dType = DPolar
E-DI3:DIndeterminacySource description = I'm not sure which function applies best? nature = Composite (Insufficient Resolution & Non Determinism & Insufficient Resolution) level = Statistical Distribution
E-DU3:DUncertainty description = What is the behavior of fresh groceries?
E-DI2:DIndeterminacySource description = we don't have information about the current rate of transmission or effectiveness of social distancing nature = Missing Information level = Recognized Ignorance
E-DU2:DUncertainty description = I don't know the relationship between "practice social distancing" and "not get or transmit COVID-19"?
E-DD3:DDecision question = Should a relationship exist? allowedPartiality = MAY resolved = False dType = DPolar
E-DR2:DInformation Requirement Dependency
E-DD4:DDecision question = Which contribution link is most appropriate? allowedPartiality = MAY resolved = False dType = DClosedEnded
E-DD7:DDecision question = Which Evolving Function describes fresh groceries the best? allowedPartiality = MAY resolved = False dType = DClosedEnded
E-DR4:DInformation Requirement Dependency
E-DD5:DDecision question = Should we decompose "Fresh Groceries" into "Pantry Staples" and "Produce" allowedPartiality = MAY resolved = False dType = DPolar
E-DR3:DInformation Requirement Dependency
E-DD6:DDecision question = What is the initial value of fresh groceries? allowedPartiality = MAY resolved = False dType = DClosedEnded
dependsOn
or
+
++
-

Fig. 5.4. The resulting mega-model after expressing the modelling uncertainties with DRUIDE on Emma's goal model

Third, the Smith researchers are uncertain about the behaviour of the *fresh groceries resource*. They model their uncertainty (*E-DU3*), localize it, characterize its *DIndeterminacySource* (*E-DI3*), articulate their decisions (*E-DD5*, *E-DD6* and *E-DD7*) and express the dependencies between them (*E-DR3* and *E-DR4*).

Forth, the modellers are having second thoughts about the goal *not get or transmit COVID-19*. In fact, they are not sure that the factors that may prevent a person from getting the virus, are as well efficient for preventing it from transmitting it. Besides, it is possible that not transmitting the virus contributes differently in reducing the spread of COVID-19, from the contribution of not getting the virus. Intuitively, it may contribute more strongly. However, there is no official information yet that clarifies this difference. For all these reasons, the modellers consider decomposing the goal *not get or transmit COVID-19* into two goals *not get COVID-19* and *not transmit COVID-19*. They express this using the objects *E-DU4*, *E-DI4* and *E-DD8*

Fifth, the modellers are uncertain about how to model the impact of wearing a mask (*E-DU5*). Specifically, they are uncertain about how to represent wearing a mask. Since there is only a limited number of possible deigns for wearing a mask (constrained by the number of possible elements in a goal model), this is a *ScenarioUncertainty* (*E-DI5*). The researchers localize this uncertainty at different parts of the model, as shown by the multiple *Localization* traces connected to *E-DU5* and represented in blue in Fig 5.4.

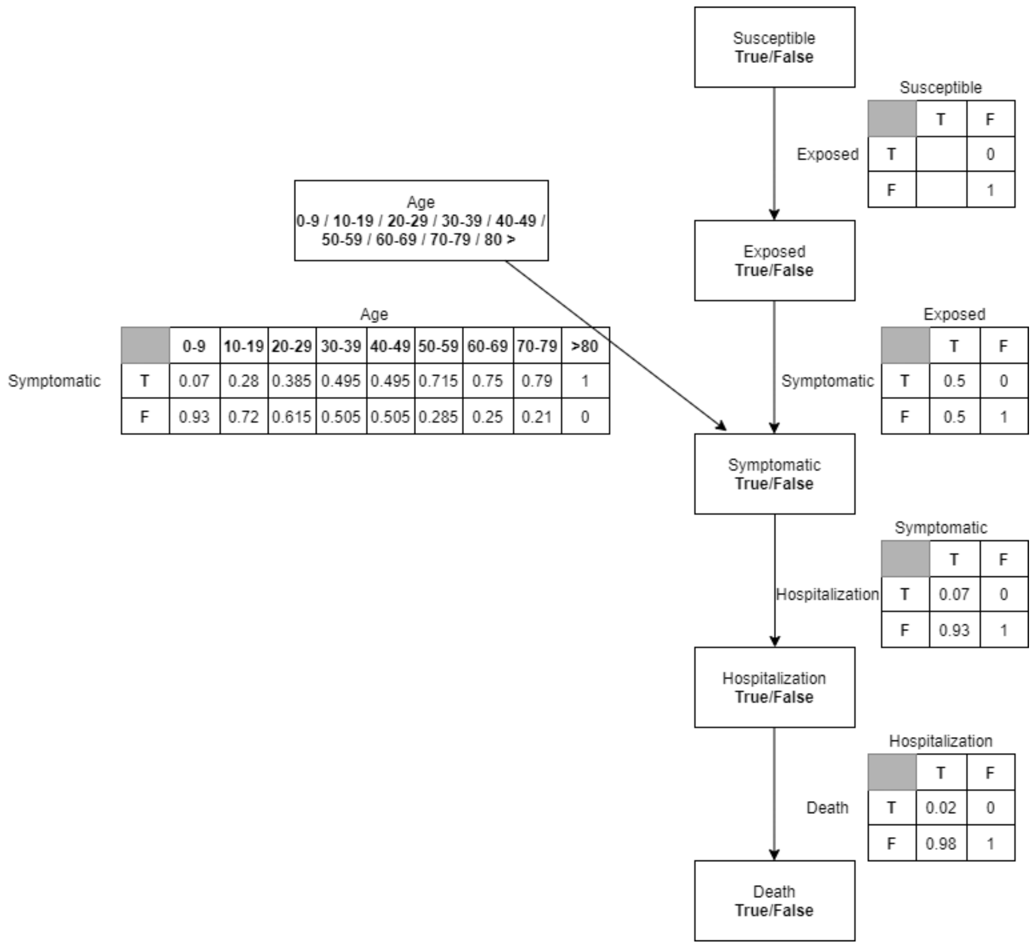
Afterwards, the modellers start articulating their decisions in order to resolve the uncertainty. Initially, they consider adding an intention for wearing masks (*E-DD9*). If they choose to do so, they need to decide what type of intention to use (*E-DD10*). Once that figured out, they need to decide the relationships that should to be connected to this intention, in order to represent the impacts of wearing a mask (*E-DD11*). In order to express this chain of dependencies, the modellers use the *DInformationRequirementDependency*.

Sixth, the researchers are unsure about how to evaluate the soft goal *minimize the spread of COVID-19*. Thus, they elaborate their uncertainty (*E-DU6*) and articulate their decision concerning this (*E-DD12*).

To conclude, using DRUIDE, the Smith researchers were able to model their inherent uncertainties concerning the Emma model, and concretize them into decisions that can be acted on. Specifically, they were able to describe the uncertainty about the model, as opposed to the uncertainty the actors may have about their decisions. Also, we note that there is no much *Operationalization* done. This is mainly because the Smith researchers struggled with that concept. We will detail this later in section 5.4.

5.3.2. Quebec's COVID-19 epidemiological model

In this real-life case evaluation, we use the report entitled *Epidemiology and modelling of the evolution of COVID-19 in Quebec* published by the Government of Quebec[5]. We focus on pages 29-33 and try to model a Bayesian Network capable of capturing the impact of respecting the lock-down measures on the probabilities of exposure to the virus, hospitalization and death. Based on the report, we are able to model the following portion of the model, presented in Fig 5.5.



Symptomatic / Age	0-9	10-19	20-29	30-39	40-49	50-59	60-69	70-79	>80
T	0.07	0.28	0.385	0.495	0.495	0.715	0.75	0.79	1
F	0.93	0.72	0.615	0.505	0.505	0.285	0.25	0.21	0

Exposed / Susceptible	T	F
T		0
F		1

Symptomatic / Exposed	T	F
T	0.5	0
F	0.5	1

Hospitalization / Symptomatic	T	F
T	0.07	0
F	0.93	1

Death / Hospitalization	T	F
T	0.02	0
F	0.98	1

Fig. 5.5. Bayesian Network of the Quebec COVID-19 Epidemiological model

The model in 5.5 presents five Boolean variables that represent the possible health states of an individual during the pandemic: *Susceptible*, *Exposed*. *Symptomatic*, *Hospitalized* and *Dead*. The transition from one state to another is estimated based on conditional probabilities shown in matrices next to the edges in the figure. We deduced these probabilities from the report. For instance, the probability of a person to be hospitalized known that he is

[5]Available here: `http://www.marc-brisson.net/covid19-response/Epidemiologie-et-modelisation-evolution-COVID-19-au-Quebec_7-mai.pdf`. URL accessed on September 13th, 2020.

symptomatic is equal to 0.07. The report also specifies the *Symptomatic* probabilities based on the age of a person. Thus, we add the *age* variable and the corresponding conditional probabilities in our model.

When modelling, we faced several uncertainties. We used Druide to express them. The resulting model is presented in Fig 5.6.

First, the report does not specify what is the probability of being exposed to the virus. We model this as the *DUncertainty DU1* with an indeterminacy source (*DI1*) of type *MissingInformation*. Then, we localize it at the Exposed/Susceptible conditional probabilities matrix.

Second, in order to map the epidemiological model to the Bayesian Network semantics, we gathered the mutually exclusive health states as the possible values for one variable. For instance, in the report, they distinguish between *Symptomatic* and *Asymptomatic*. In our bayesian model, we used only the *Symptomatic* variable with two values: True and False. When the value is False, the probabilities express the state *Asymptomatic*. Similarly, the epidemiological model has two states: *Death* and *Recovered*. In our bayesian model, we use *Death* variable with True and False values. The False value refers to the *Recovered* state. However, since we do not know what should the bayesian model emphasize, we are not sure which variable is better to use: *Death* or *Recovered*, *Symptomatic* or *Asymptomatic*. We model this uncertainty as the *DUncertainty DU2*. Then, we characterize its indeterminacy source. Specifically, we have two alternatives for each variable, so we are at the *ScenarioUncertainty* level (*DI2*). Besides, although we know that the purpose of this model is to help Emma make the safest decision, we don't know how this will be done yet. So, we don't precisely know what the model should emphasize and what variables should be used. We characterize this situation as *InsufficientResolution* (*DI2*). Afterwards, we localize *DU2* in the *Symptomatic* and *Death* variables as shown using the blue lines connected to *DU2* in the figure.

After that, we elaborate on our uncertainty and express two concrete *DClosedEnded* design decisions: *DD1* and *DD2*. Since, the articulation of *DD1* is similar to *DD2*, we only articulate *DD2* to reduce visual clutter. Specifically, we refine *DD2* into two mutually exclusive *DPolar* decisions (*DD3* and *DD4*). Then, we operationalize each of them. For example, the *Operationalization* of *DD3* resulted in the addition of the *Recovered* variable, the edge linking the *Hospitalization* state to the *Recovered* state and the Recovered/Hospitalization matrix. These elements are connected to *DD3* using the *Operationalization* red traces.

Third, we are not sure how to model the fact of respecting preventive measures. The report distinguishes between two measures: *Reduce contacts* and *Isolation if symptomatic*. We are uncertain whether we should gather both of them as one variable, or distinguish between the two. So, similarly, we model this as an uncertainty (*DU3*), we characterize its indeterminacy source (*DI3*), then we articulate and evolve its related decisions (*DD5*, *DD6* and *DD7*). Finally, we operationlize the *DPolar* ones (*DD6* and *DD7*).

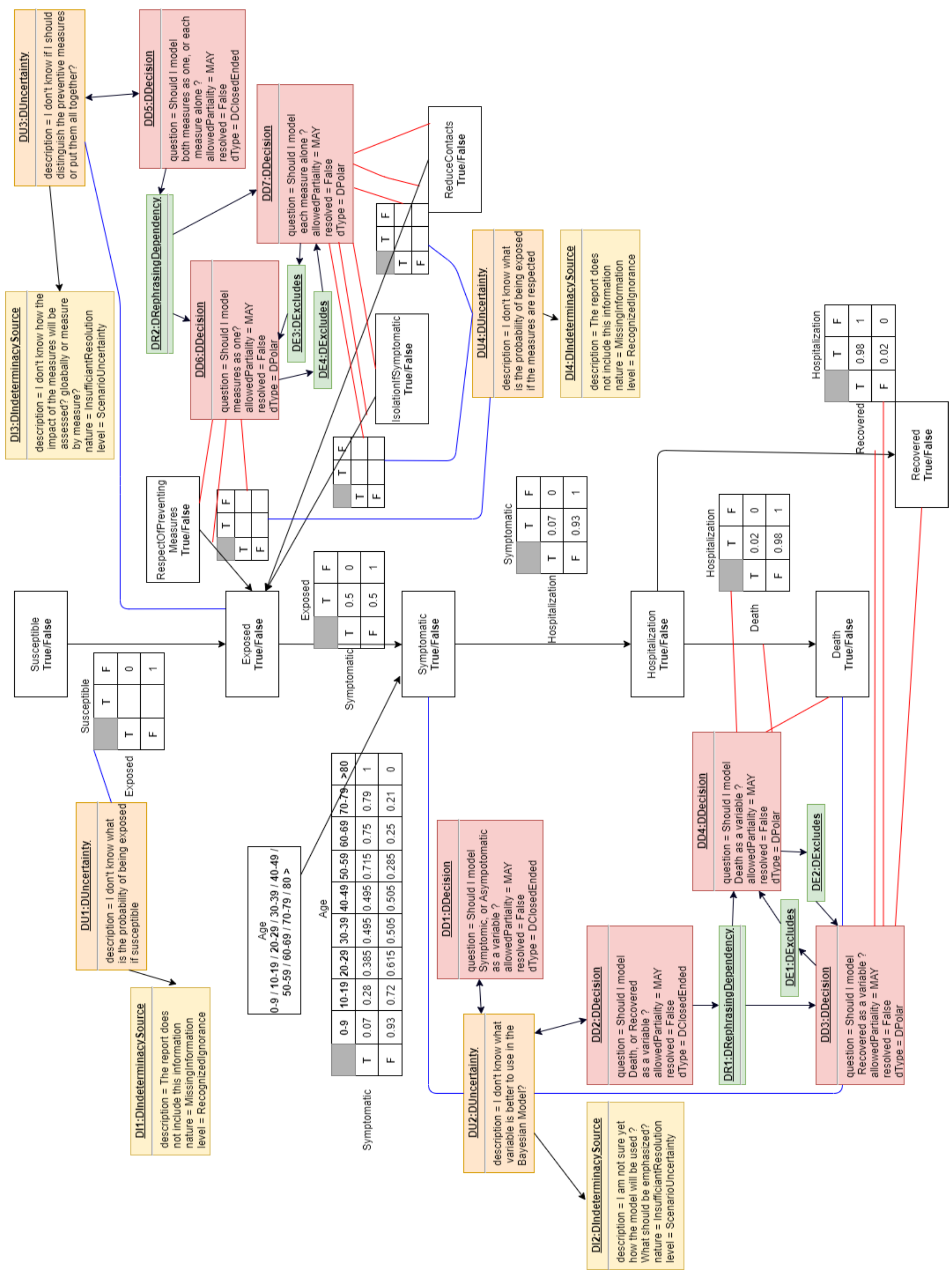

Susceptible True/False
Exposed True/False
Symptomatic True/False
Hospitalization True/False
Death True/False
Recovered True/False
RespectOfPreventing Measures True/False
IsolationIfSymptomatic True/False
ReduceContacts True/False
Age 0-9 / 10-19 / 20-29 / 30-39 / 40-49 / 50-59 / 60-69 / 70-79 / 80 >
DU1:DUncertainty description = I don't know what is the probability of being exposed if susceptible
DI1:DIndeterminacySource description = The report does not include this information nature = MissingInformation level = RecognizedIgnorance
DU2:DUncertainty description = I don't know what variable is better to use in the Bayesian Model?
DI2:DIndeterminacySource description = I am not sure yet how the model will be used ? What should be emphasized? nature = InsufficiantResolution level = ScenarioUncertainty
DU3:DUncertainty description = I don't know if I should distinguish the preventive measures or put them all together?
DI3:DIndeterminacySource description = I don't know how the impact of the measures will be assessed? globally or measure by measure? nature = InsufficiantResolution level = ScenarioUncertainty
DU4:DUncertainty description = I don't know what is the probability of being exposed if the measures are respected
DI4:DIndeterminacySource description = The report does not include this information nature = MissingInformation level = RecognizedIgnorance
DD1:DDecision question = Should I model Symptomic, or Asympotomatic as a variable ? allowedPartiality = MAY resolved = False dType = DClosedEnded
DD2:DDecision question = Should I model Death, or Recovered as a variable ? allowedPartiality = MAY resolved = False dType = DClosedEnded
DD3:DDecision question = Should I model Recovered as a variable ? allowedPartiality = MAY resolved = False dType = DPolar
DD4:DDecision question = Should I model Death as a variable ? allowedPartiality = MAY resolved = False dType = DPolar
DD5:DDecision question = Should I model both measures as one, or each measure alone ? allowedPartiality = MAY resolved = False dType = DClosedEnded
DD6:DDecision question = Should I model measures as one? allowedPartiality = MAY resolved = False dType = DPolar
DD7:DDecision question = Should I model each measure alone ? allowedPartiality = MAY resolved = False dType = DPolar
DR1:DRephrasingDependency
DR2:DRephrasingDependency
DE1:DExcludes
DE2:DExcludes
DE3:DExcludes
DE4:DExcludes

Fig. 5.6. The resulting mega-model after applying DRUIDE on the Bayesian Network of Quebec's COVID-19 epidemiological model

Finally, in both scenarios, the report didn't include the impact of respecting the preventive measures on being exposed to the virus. Thus, we express and localize this uncertainty (*DU4*).

In this example, we have shown that, thanks to DRUIDE, we were able to articulate all of our uncertainties. It is also worth noting that Fig 5.6 depicts uncertainty at two different levels: the bayesian belief network describes the uncertainty contained inside the model, while DRUIDE is used to articulate uncertainty about the model.

After successfully applying DRUIDE on two real-life evaluation scenarios (Emma's goal model and the bayesian network model of the Quebec COVID-19 epidemiological model), we can answer **RQ2-2**.

RQ2-2 results: DRUIDE *is suitable for real-life scenarios.*

Besides, after successfully applying DRUIDE on five intra-model uncertainties (two real-life evaluation scenarios and three literature-based evaluation examples), we can answer **RQ2-3**.

RQ2-3 results: DRUIDE *can express intra-model uncertainties.*

In the next section, we address **RQ2-4**.

5.3.3. Linking Emma's model with Quebec's epidemiological model

After finishing modelling Emma's goal model and the Quebec's epidemiological model separately, we try to link both pieces of the case study together to have the complete picture. In fact, in order to help Emma make the right choice, we consider the potential impact of her decisions on the Bayesian network variables. Specifically, we map the goal model tasks to the corresponding evaluation of the Bayesian variables they might impact. For instance, when Emma chooses to cook at home, it is logical to stay that she will be respecting the preventive measures by reducing contact. In other words, in case Emma chooses to cook at home, it will be similar as evaluating the Bayesian variables *Reduce contacts* and *Respect of Preventing Measures* to *True*. We use the two dark purple links linked to the *Cook at home* task to denote this mapping as shown in Fig 5.7. We also note that we omit several DRUIDE elements to reduce visual clutter in Fig 5.7.

We emphasize the fact that these lines do not correspond to any objects, classes or models. Instead, they only refer to some background mechanism that assigns a specific value

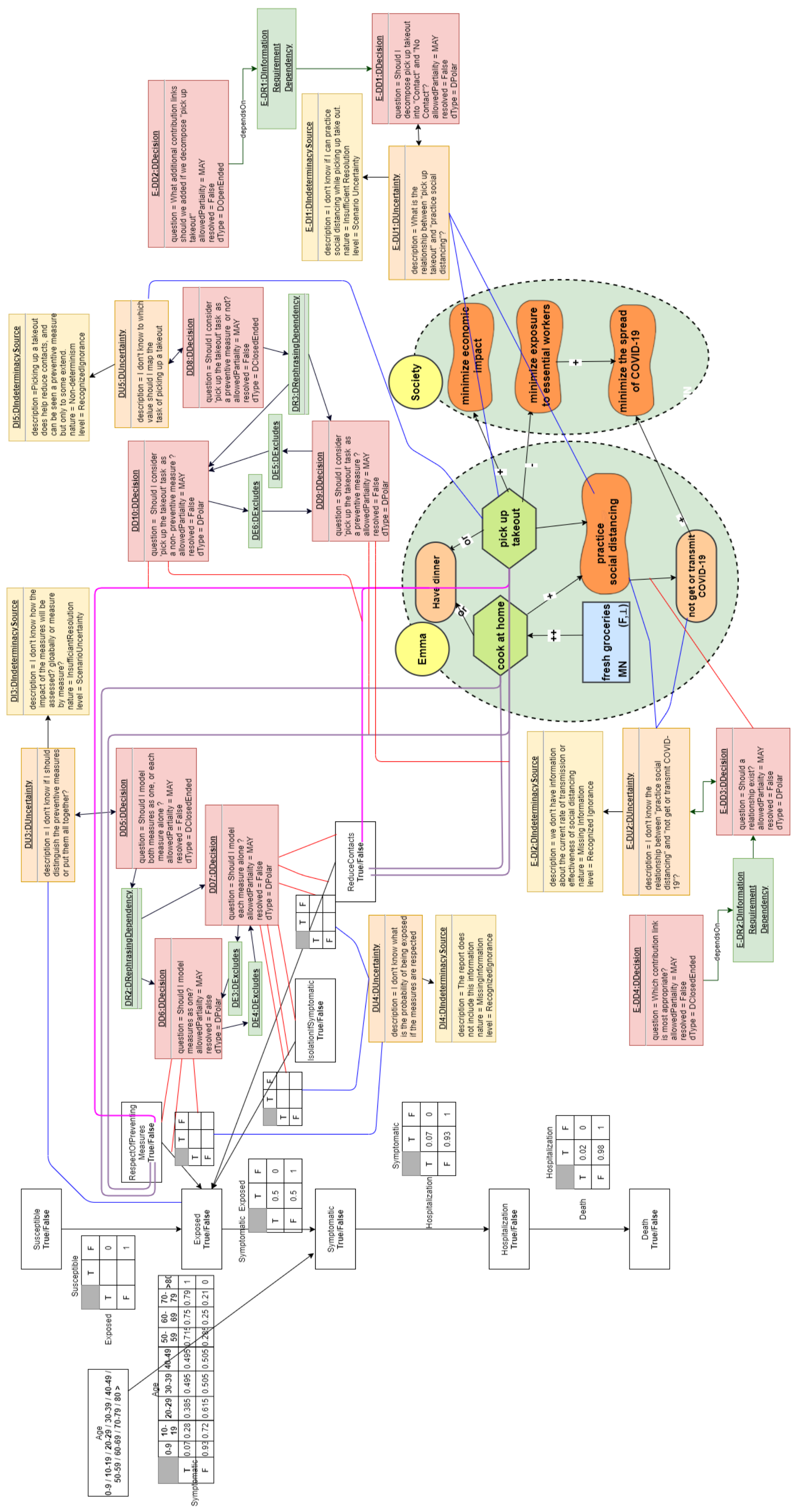


Fig. 5.7. Druide on the mapping

to a variable, whenever a certain a task has been made, thus triggering the update of the bayesian network algorithm.

When we try to map the other task, *pick up takeout*, to the corresponding variables values, we are faced with uncertainty; Does picking up a takeout break the preventive measures, or does it respect them? Does it help reducing the contacts, or not? Intuitively, it is definitely better than eating outside, for example, at a restaurant, but it is riskier than cooking at home. Thus, we are uncertain if we should consider it as a preventive measure, or as a non-preventive measure.

Since we are unsure about the mapping, we use DRUIDE to represent this uncertainty (*DI5, DU5*) and articulate its corresponding decisions (*DD8, DD9, DD10*). We operationalize each of the alternatives as follows. *DD10* means we consider picking up a takeout as a non-preventive measure, and thus it is mapped to the *False* values of the *Reduce contacts* and *Respect of Preventing Measures* variables (Two bright pink links linked to the *pick up takeout* task in Fig 5.7). On the other side, operationalizing *DD9* results in mapping the *pick up takeout* task to the *True* values of the *Reduce contacts* and *Respect of Preventing Measures* variables (Two dark purple links linked to the *pick up takeout* task in Fig 5.7). In the latter case, it is the same evaluation as for the *cook at home* task. This means, that in case we choose to make *DD9* and consider that picking a takeout is a preventive measure, the bayesian network will indicate that both tasks are equivalent. In fact, the probability of exposure will be the same, meaning that both tasks will result in the same risk.

When merging both models into one, we noted similar uncertainties. Specifically, the *E-DU2* uncertainty that concerns the relationship between practicing social distancing and not getting or transmitting the virus in Emma goal model, is the same as the *DU-4* uncertainty that concerns the impact of respecting preventive measures on the probability of exposure to the virus in the bayesian network. Although rephrased differently (since they have been articulated by different modellers), they both refer to the same uncertainty, which is the effectiveness of the preventive measures on reducing the spread of the virus. Similarly, the *E-DU1* uncertainty that concerns the relationship between picking up a takeout and practicing social distancing in Emma's goal model, refers to the *DU5* mapping uncertainty explained above. These are illustrations of how the same uncertainty can appear in different models.

Finally, we note that the merge of the two models, partly presented in Fig 5.7, depicts three different types of uncertainties:

- Emma uncertainty concerning her decision to have dinner captured by the goal model.
- The uncertainty concerning the Quebec COVID-19 epidemiological model expressed using the Bayesian network probabilities.

- The uncertainty concerning the models and the relationships between them articulated using Druide.

Thus, this case study clearly illustrates that different uncertainty types require different modelling approaches and different treatments. Existing modelling notations, like Bayesian networks or goal models, are suitable for representing uncertainty inside the system, however they do not cover the uncertainty about the design of the system. Druide, on the other side, covers exactly that type of uncertainty.

The main purpose of this part was to evaluate Druide ability to express inter-model uncertainties. We have demonstrated this ability using the simple representative case introduced above from two perspectives:

(1) Uncertainty can appear in the relationships between models. We have already successfully used Druide to articulate uncertainties concerning the relationships between two different models.
(2) The same uncertainty can appear in different models. We have already noted similar uncertainties above. In Druide, these can be easily merged into one, and have associated decisions in different models.

Thus, we can answer **RQ2-4**.

RQ2-4 results: Druide *can express inter-model uncertainties.*

To conclude, after conducting both the lab-based evaluation and the case study, we can answer **RQ2**.

RQ2 results: Druide *is expressive enough for representing both intra-model and inter-model uncertainties, for both in-lab and real-life scenarios.*

In the next section, we address **RQ3**.

5.4. Post-study questionnaire

In order to collect the opinions of our three participants on Druide after their modelling experience, and to answer RQ3 concerning the usability of Druide, we used a post-study questionnaire. The questionnaire we proposed contains four questions as listed below:

- **Q1:** To what extent do you find Druide adequate for uncertainty-aware modelling?
- **Q2:** Describe your experience when modelling your uncertainties about the Emma example using Druide. Was it expressive enough for you? Were there any uncertainties that you could not model with Druide?

- **Q3:** Describe your process of learning DRUIDE. Explain the challenges you faced. Do you think that the time needed was reasonable, with respect to the benefits? Were there any concepts particularly hard to understand? If yes, please name them. Did you need additional resources in order to be able to use DRUIDE adequately? If yes, please discuss them and say how they helped you.
- **Q4:** After one modelling experience, to what extent do you think you can use DRUIDE concepts, without any assistance or doubts? According to you, how much time or modelling experience is approximately needed to master DRUIDE concepts?

The first question **Q1** concerns the first research question **RQ1: Adequacy**, while the second one **Q2** relates to the second research question **RQ2: Expressiveness**. Finally, we asked the third and forth questions; **Q3** and **Q4**, to evaluate the third research question **RQ3: Usability**. The rest of this section is structured by research question. In each subsection, we present the participants responses and comment on them.

5.4.1. RQ1: Adequacy.

In this subsection, we focus on the answers we received from the participants concerning the first question **Q1**. The three answers are listed below.

- **Student1**: *Uncertainty plays a large role in modelling and I think its failure to be articulated adds a barrier to real-world utility. Uncertainty-aware modelling addresses this barrier and I think it's more truthful and brings the model closer to the real-life situation. I see it (*DRUIDE*) as highly adequate for those reasons.*
- **Student2**: *I found* DRUIDE *adequate for the modelling that I did. (...)*
- **Professor**: *I found* DRUIDE *completely adequate for uncertainty-aware modelling. Once the additional dependency was added (the DInformationRequirementDependency), I was able to connect elements with appropriate dependency links.*

Although the first answer showed the utility of any uncertainty-aware modelling language in general, the second two insisted on DRUIDE adequacy for uncertainty-aware modelling.

RQ1 results: *The participants answers confirm the evaluation results.*

5.4.2. RQ2: Expressiveness.

In this subsection, we focus on the answers we received from the participants concerning the second question **Q2**. The three answers are listed below.

- **Student1**: *I think every uncertainty we had with Emma could be articulated in some form using* DRUIDE *(...). The* DRUIDE *modelling process itself also helped discover more uncertainties in the model because of the mindset it provokes.*
- **Student2**: *Because any uncertainty could be expressed as a DUncertintanty, I had no issue modelling all uncertainties in* DRUIDE*. (...)*
- **Professor**: *We started by creating the base goal model of Emma. Then we brainstormed about the uncertainties we experienced while modelling. (...) All the uncertainties that we brainstormed, we were able to model in some form. Upon review, Mouna corrected our InformationRequirementsDependency. (...)*

The three participants agreed that they were able to express all of their uncertainties concerning Emma's goal model using DRUIDE.

RQ2 results: *The participants answers confirm the evaluation results.*

5.4.3. RQ3: Usability.

In this subsection, we focus on the answers we received from the participants concerning the third and forth questions **Q3** and **Q4**. The three answers for **Q3** are listed below.

- **Student1**: *When initially modelling with* DRUIDE*, it was difficult to know where to start with no prior experience, and felt a bit intimidating. I was especially confused on identifying the DIndeterminacyNature and still struggle with that component. (...)*
- **Student2**: *I learned* DRUIDE *in several short bursts, while working on the Emma case study. The basics of adding uncertainties was not very hard, but there were some fundamental misunderstandings that my modelling team had, which went uncorrected until we met with more experienced modellers. (...)*
- **Professor**: *We reviewed the meta-model, it was very helpful with the explanation provided by Mouna/Michalis. (...) We had difficulty with the attributes of the IndeterminacySource object, as well as allowedPartiality in the Decision object. . . .and the operationalization. The time needed to understand the concepts was reasonable. It would also be helpful to explain the difference between uncertainty in the model and uncertainty about the model. Once we got this idea straight it was easier to use* DRUIDE *and learn the concepts.*

The three researchers agreed that using DRUIDE was not intuitive at first, that it needed time and that some concepts were harder to master than others.

The three answers for **Q4** are listed below.

- **Student1**: *After one modelling experience, I feel familiar with* Druide *concepts. While I think I could interpret a* Druide *model somewhat comfortably if presented to me, I would struggle with adding* Druide *on top of a model myself, especially without a guide. Mastering* Druide *is definitely a matter of practice, and I would estimate at least five more guided modelling activities are needed for mastery.*
- **Student2**: *After this experience, I feel like I understand* Druide *on a surface level, and enough to use it in another similar situation, but I would not be able to go closer. I think I could easily add DUncertainties to a model and localize them, and would probably be able to add some DDescisions and DIndeterminancySources but there would be errors in those additions. I would probably have to work on one project at this level and at least one more advanced modelling project to be fully versed in* Druide *concepts.*
- **Professor**: *I think I could use the main* Druide *concepts without any help, but I still need help on allowedPartiality in the DDecision object and the operationalization. I think this is more trial and error. I think I would require another afternoon of modelling to master* Druide *concepts.*

The three researchers agreed that one modelling experience is not enough to master Druide concepts, and that more practice is needed.

After conducting the post-study questionnaire and analyzing the participants answers regarding questions **Q3** and **Q4**, we can answer **RQ3**.

RQ3 results: Druide *is usable but mastering it requires time and practice.*

Finally, we asked the Smith researchers about their feedback concerning the usage of the Draw.IO tool[6] that we used to create the object diagrams presented above. The answers we received were:

(1) *It did feel a bit repetitive after articulating a few uncertainties, but that could mainly be attributed to using Draw.IO, and having to manually add links/etc.*

(2) *We found the numbering (identification) of elements very tedious and it was difficult to make sure that we didn't double use the numbers.*

In order to overcome Draw.Io's usage inconveniences, we propose a tool that supports Druide in the next chapter.

[6]Draw.IO is a tool for drawing diagrams. It can be downloaded from here: `https://github.com/jgraph/drawio-desktop/releases`. URL accessed on September 13th, 2020.

5.5. Threats to validity

In this section, we present the threats to validity of our evaluation. Internal validity is concerned with our choice of evaluation systems. In fact, the three worked examples from the literature as well as the COVID-19 real-world case study can be considered as extremely suitable cases, with characteristics perfectly tailored for our language. Another internal threat is the maturation threat, that potentially happened as a result of the natural maturation of the Smith modellers as they were learning DRUIDE.

External validity refers to the generalizeability of our findings. Our evaluation concerned a limited number of systems (3 from the literature and 1 from real life) and involved a total of 3 participants. Thus, we cannot assert that our results can be generalized to other systems, and other modellers. Therefore, more evaluation is necessary to confirm our findings. Finally, there is a possible threat due to experimenter bias in the post-questionnaire answers as the subjects had prior contact with the researchers.

Construct validity is concerned with the relationship between theory and what is observed. Our evaluation might be assessing the modellers ability to express and articulate their uncertainties, rather than DRUIDE ability to model the real uncertainties. Besides, another construct validity threat concerns the subjective understanding of DRUIDE concepts.

In this chapter, we presented the field-based evaluation of DRUIDE. In the next chapter, we show DRUIDE feasibility by introducing a tool that supports our language, implements our operators and encodes our constraints.

Chapter 6

Tool Support

In this chapter, we report the two approaches we have investigated in order to provide tool support for our proposed language and methodology, The first one, which was our initial attempt, is based on meta-model evolution, while the second one is grounded on native support for mega-models. We also discuss the limitations of both tools.

6.1. The *AddUncertainty* Workflow

In this section, we present and discuss our proposed tool to create an automated meta-model based approach that transforms any modelling language to an uncertainty-aware version. This is a prerequisite for the creation of tooling that allows modellers to express their uncertainties and corresponding design decisions.

We focus specifically on the feasibility of providing tool support for graphical domain-specific languages (DSL) by implementing a prototype based on AToMPM, a web-based modelling environment [**37**]. Therefore, we only consider a simplified uncertainty meta-model. The meta-model is illustrated in Figure 6.1. It consists of an *Uncertainty* element to which is related a set of design *Decisions*. Our tool, the *AddUncertainty* workflow, automatically extends a given meta-model and modelling environment with that simplified uncertainty meta-model and the traces meta-model (Fig 3.3).

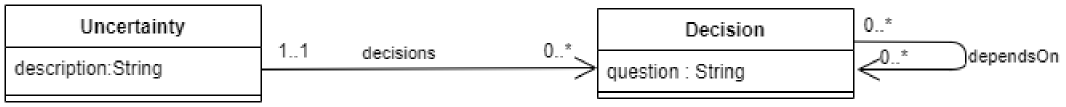


Fig. 6.1. Simplified uncertainty meta-model

To implement our approach, we use the AToMPM workflow engine [**19**]. Specifically, we propose an automated workflow, shown in Figure 6.2, that takes as input any graphical DSL and produces as output the DSL extended with uncertainty. The resulting language incorporates the input language and our uncertainty sub-language. Model transformations in

AToMPM are capable to connect elements from different meta-models. We use this technique to add the *Localization* and *Operationalization* traces as two possible associations between the input language classes and our uncertainty language classes, in the resulting meta-model.

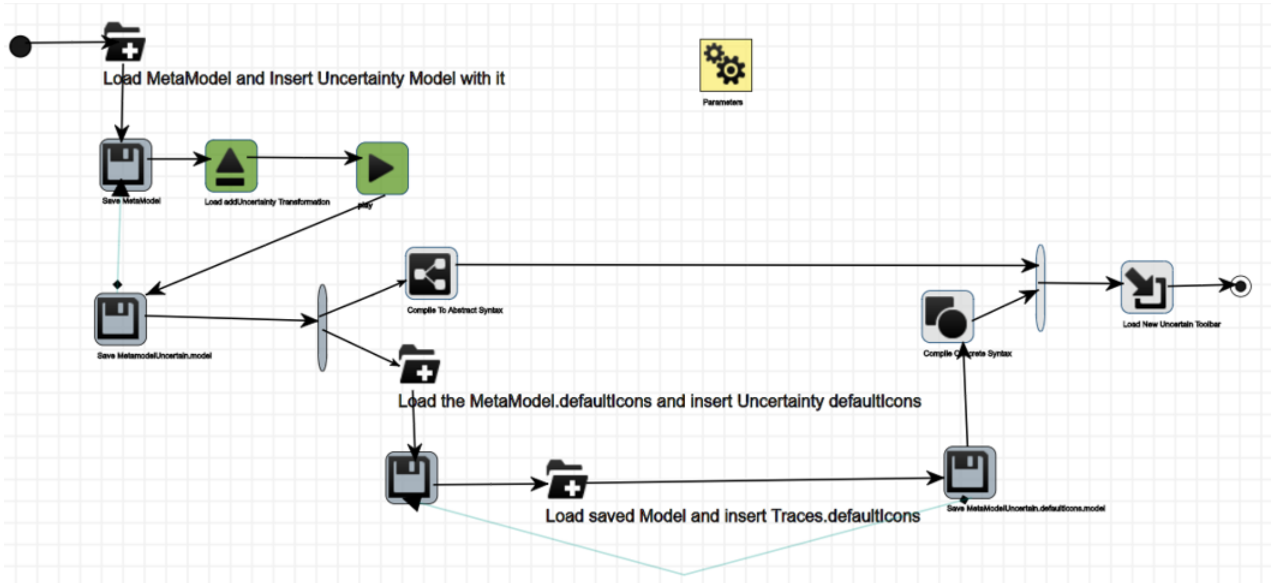

Load MetaModel and Insert Uncertainty Model with it
Parameters
Save MetaModel
Load addUncertainty Transformation
play
Save MetamodelUncertain.model
Compile To Abstract Syntax
Load the MetaModel.defaultIcons and insert Uncertainty defaultIcons
Compile Concrete Syntax
Load New Uncertain Toolbar
Load saved Model and insert Traces.defaultIcons
Save MetaModelUncertain.defaultIcons.model

Fig. 6.2. AddUncertainty workflow in AToMPM

More specifically, the workflow takes the input meta-model and inserts our sub-language next to it. It then runs the *AddUncertainty* transformation to create the traces associations between the two models. The result of the transformation is afterwards saved as the new "Uncertain Meta-model", which corresponds to the new language's abstract syntax. In AToMPM, in order to use a language, we need to compile both its abstract and concrete syntaxes. This is exactly what the workflow does simultaneously. For the abstract syntax, it simply compiles it. Concerning the concrete syntax, it starts by putting the original meta-model concrete syntax, our sub-language concrete syntax and the traces concrete syntax side by side, then it saves the whole as the "Uncertain Meta-model" concrete syntax. After that, it compiles it. Once both the abstract and concrete syntaxes are compiled, the workflow loads the new language's toolbar enabling the user to model his system.

In order to highlight better the impact of our automated tool on the modelling environment, we re-consider the Peer-to-peer example, presented in section 4.2.2. We assume the modellers are using a simplified state machine meta-model provided in the AToMPM modelling tool to model the behaviour of the peer-to-peer system. Figure 6.3 shows the current state of modelling.

The yellow-framed palette at the top left of Figure 6.3 exposes to the modeller the available modelling concepts from the original state machine meta-model. When modelling the peer-to-peer system, the engineers can only model a *State*, a *Transition*, an *Action*, and

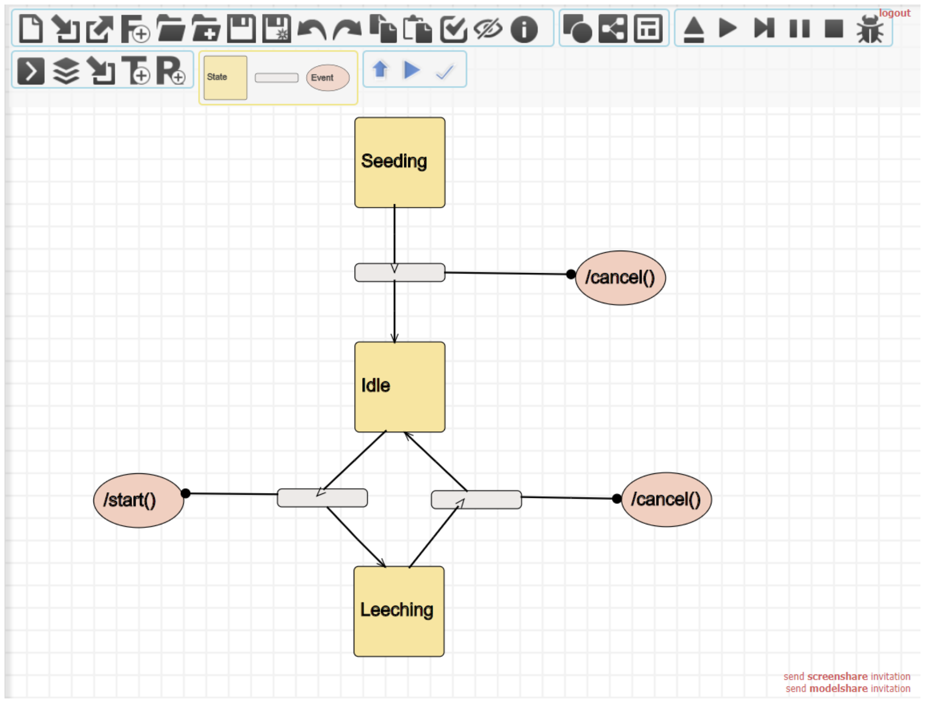


Fig. 6.3. The unfinished model of the peer-to-peer system modeled in AToMPM.

the permitted associations between them as defined in the original state machine meta-model: in other words, they do not have at their disposal any facilities to express their uncertainty.

Applying the *AddUncertainty* workflow on the state machine meta-model updates the language to a new one that includes the initial state machines meta-model, as well as the uncertainty meta-model and the traces. The result of the workflow can be seen in the yellow-framed menu at the top right of Figure 6.4, which now also exposes modelling concepts from the uncertainty sub-language. Thus, the modellers of the peer-to-peer example are now able to express their uncertainties and model their decisions.

In Figure 6.4, the modellers' expressions of uncertainty are shown as orange boxes and their decisions as pink ovals. Specifically, the modellers have expressed their uncertainties about the behaviour of the system as two *Uncertainty* elements. They also represented the decisions *D1-D3* (section 4.2.2) as *Decision* elements linking them to the corresponding *Uncertainty* elements.

Furthermore, the modellers were able to localize their uncertainties in the corresponding parts of the system. In Figure 6.4, the *Localization* traces are shown as green links. For instance, the *Uncertainty* element "uncertainty about the seeding state" is localized in the *Seeding* state. The modellers also used the *Operationalization* traces, shown as blue links, to show how the elicited design decisions affect the model. In the example, the modellers chose to operationalize the first and second design decisions by adding two transitions with the respective actions *share()* and *restart()*, as explained in section 4.2.2.

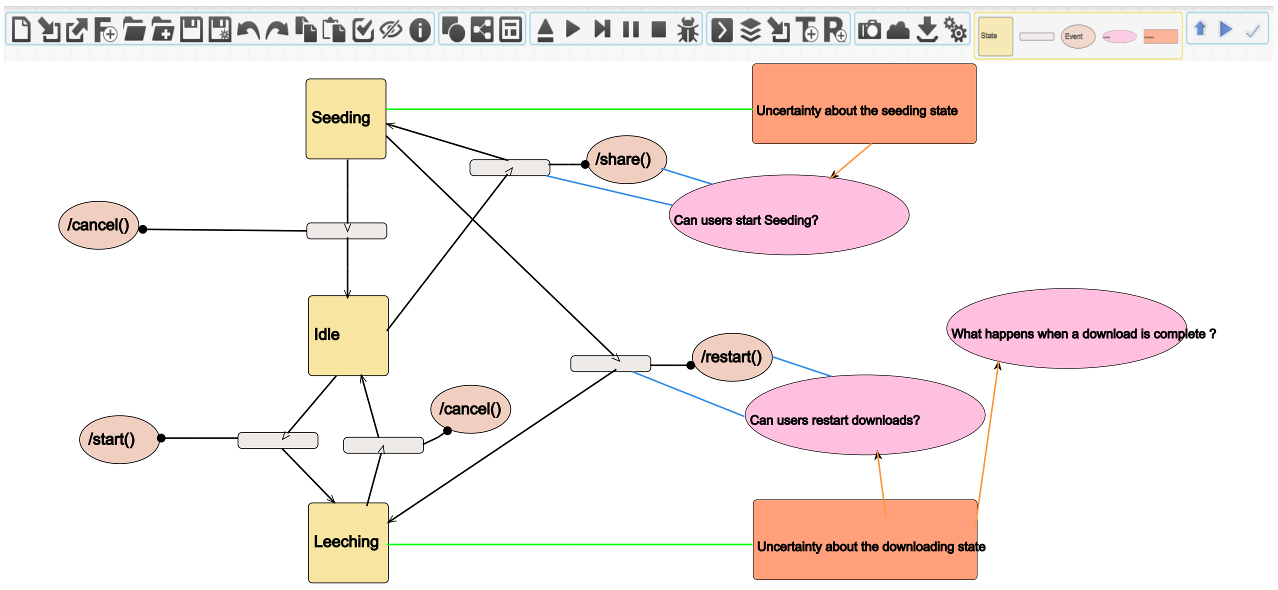

State
Event
Seeding
Uncertainty about the seeding state
/share()
Can users start Seeding?
/cancel()
Idle
What happens when a download is complete ?
/restart()
Can users restart downloads?
/cancel()
/start()
Leeching
Uncertainty about the downloading state

Fig. 6.4. Articulation of design uncertainty for the peer-to-peer example using the uncertainty-aware modelling language for state machines generated with the *addUncertainty* workflow.

This prototype exposed AToMPM's limitations; mainly in terms of support for meta-model co-evolution and for mega-modelling.

First, the language extension performed by the *AddUncertainty* workflow does not alter the classes and associations defined in the input meta-model. It simply inserts the uncertainty sub-language and connects the corresponding classes with the appropriate associations. So, in its current form, the workflow does not affect the original meta-model and its constraints. Thus, in principle, the resulting meta-model can be used to type instances conforming to the original meta-model, i.e., instances of the original meta-model are also instances of the new meta-model. However, in practice, due to technical limitations of AToMPM, this is not the case in our current implementation. For instance, we have to copy the original meta-model which changes the namespace of the classes.

Second, since AToMPM does not provide explicit support for mega-models and traces, we introduced a couple of workarounds (adding attributes, transformations, abstracts classes...) in order to make its concepts fit our purposes. For instance, we adapted our approach so that all three meta-models (original meta-model, uncertainty meta-model and the traces meta-model) are merged into one. Thus, practically, we are no longer dealing with a mega-model, but rather simply only one meta-model and instances of it.

For these reasons, we abandoned AToMPM and investigated another tool that naturally supports mega-modelling. We detail this in the next section.

6.2. D-Mmint

In this section, we present D-Mmint, a tool that supports modelling in the presence of design-time uncertainty. D-Mmint is based on Mmint ("*Model Management INTeractive*")[1], which is an Eclipse-based graphical tool for interactive model management [**8**].

We have chosen Mmint among the available model management tools for several reasons. First, we wanted to develop our approach in the Eclipse software ecosystem. Second, Mmint natively supports mega-modelling at various meta-levels, which provides a natural way for language extension that guarantees reuse of model management operators. Third, unlike the other tools that provide programming capabilities for model management, Mmint provides an interactive graphical user environment that allows users to perform automatically assisted model management tasks [**8**]. Forth, Mmint has an integrated querying engine [**7**] that we need in order to implement the constraints. Finally, Mmint has previously been extended to support uncertainty modelling using partial models [**13**], which aligns with our future work, as we will detail later in section 8.2.1.

The D-Mmint tool supports our language, implements our operators and encodes our constraints. In the rest of this section, we will explain this and illustrate D-Mmint usage by considering the peer-to-peer example presented in section 4.2.2.

D-Mmint simply extends the Mmint type-level mega-model with our Druide model, by the installation of the Druide Eclipse plugin[2]. In order to extend a given meta-model with Druide, we have to define the *Localization* and *Operationalization* operations as two binary model relationship types between them, at the type-level. Figure 6.5 presents a screen shot of the part of the type mega-model in Mmint that shows the Druide model, the State Machines model and the *Localization* and *Operationalization* relationships between them.

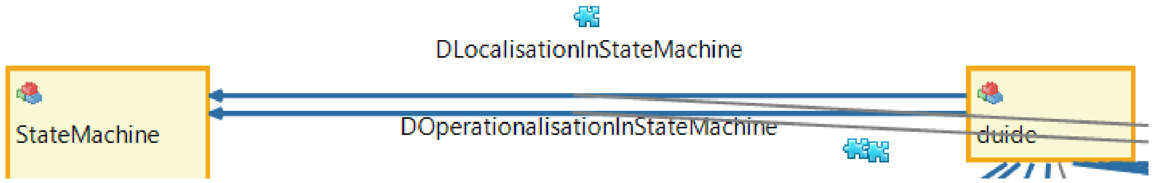

DLocalisationInStateMachine
StateMachine
DOperationalisationInStateMachine
duide

Fig. 6.5. A screen shot of a fragment of the type mega-model in D-Mmint that shows Druide model, the State Machine model and the *Localization* and *Operationalization* relationships between them.

Figures 6.6 and 6.7 respectively show the definitions of the above binary model relationships; *DLocalisationInStateMachine* and *DOperationalisationInStateMachine*, at the type

[1]Available at: `https://github.com/adisandro/MMINT`. URL accessed on September 13th, 2020.
[2]The Eclipse Update Site project is Available at: `https://udemontreal-my.sharepoint.com/:f:/g/personal/mouna_dhaouadi_umontreal_ca/Et0UQSEL_m9Eu2HXC0diogMBNz1oftCoxCdNlSz3cmpJaA?e=oF6Egm`. URL accessed on September 13th, 2020.

level. The figures show how the relationships are defined in terms of possible mappings between the models elements.

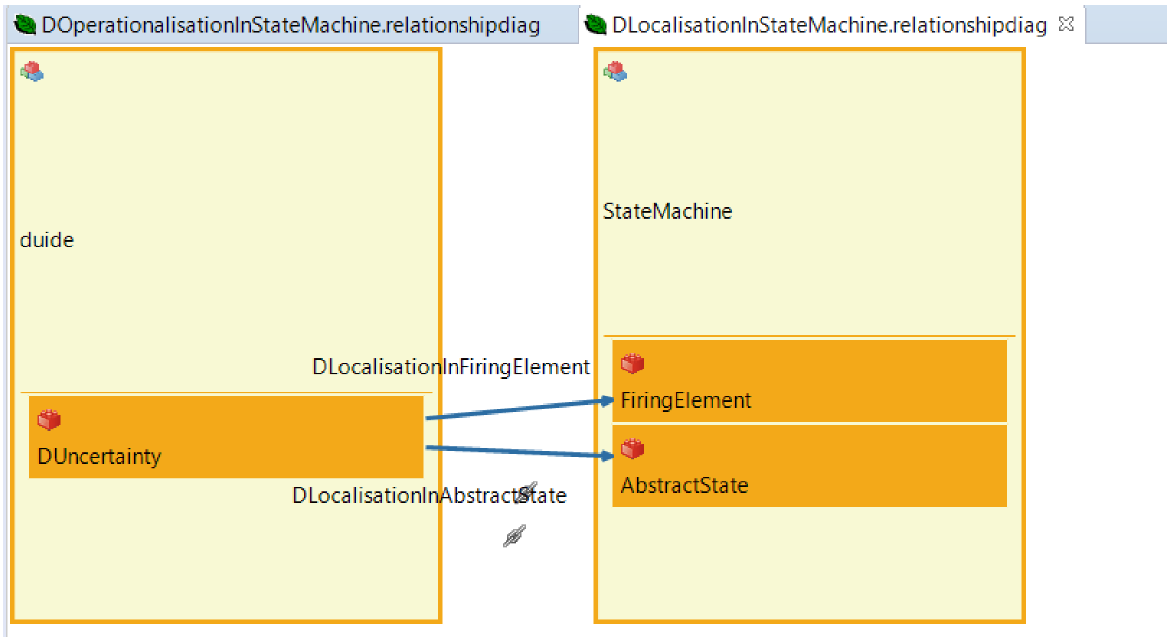

DOperationalisationInStateMachine.relationshipdiag
DLocalisationInStateMachine.relationshipdiag
duide
StateMachine
DLocalisationInFiringElement
FiringElement
DUncertainty
AbstractState
DLocalisationInAbstractState

Fig. 6.6. Definition of the *DLocalisationInStateMachine* relationship at the type level.

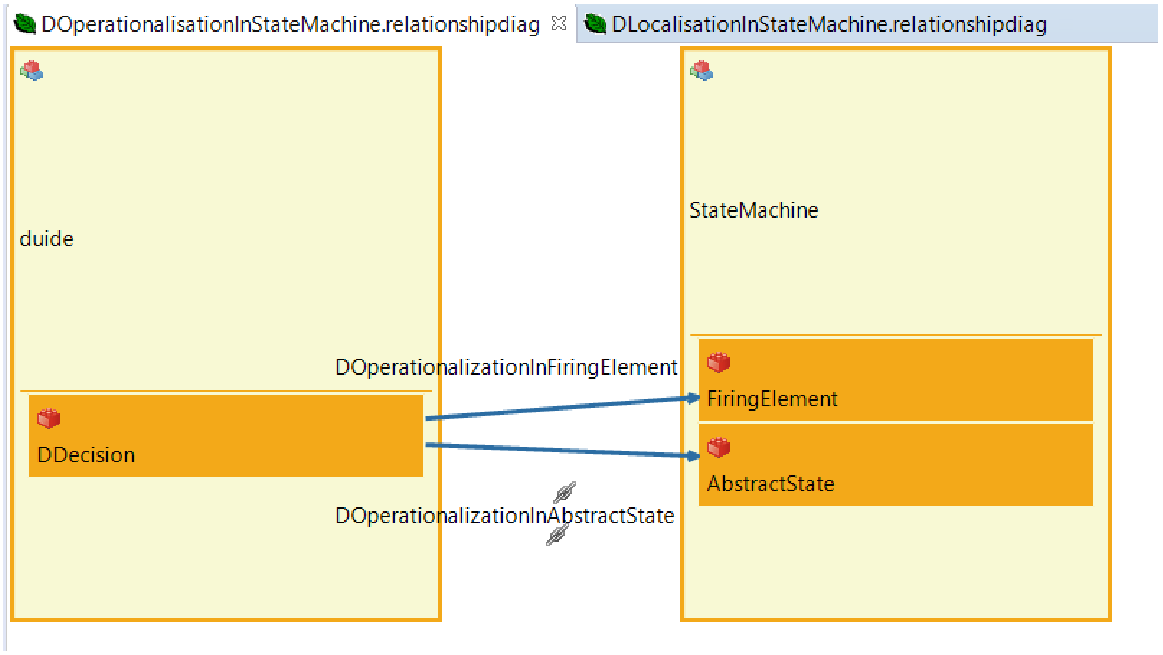
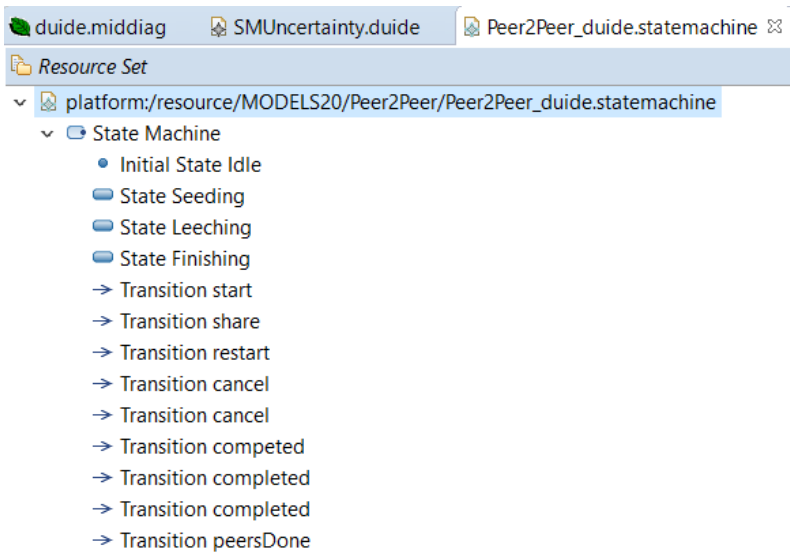

DOperationalisationInStateMachine.relationshipdiag
DLocalisationInStateMachine.relationshipdiag
duide
StateMachine
DOperationalizationInFiringElement
FiringElement
DDecision
AbstractState
DOperationalizationInAbstractState

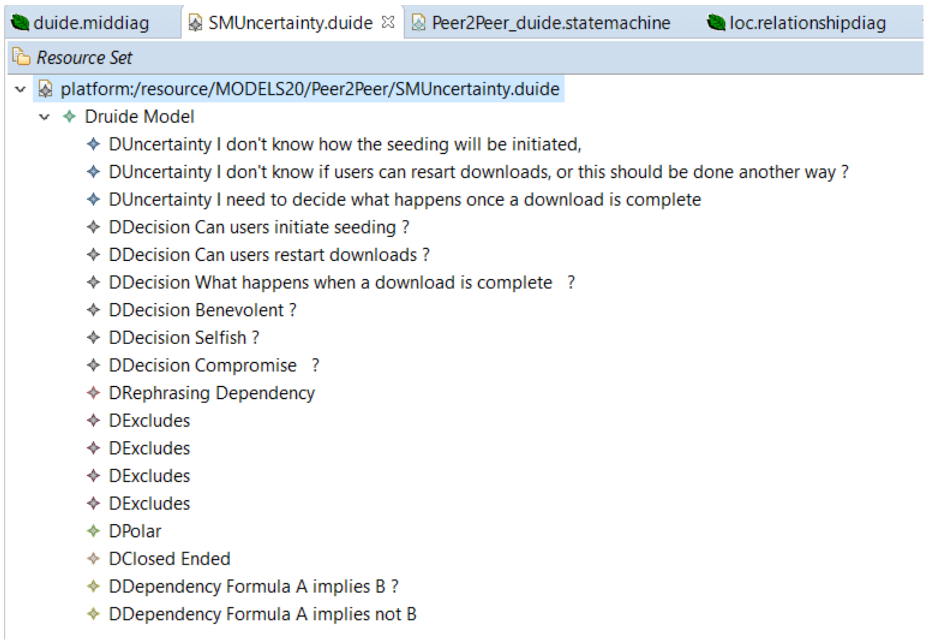
Fig. 6.7. Definition of the *DOperationalisationInStateMachine* relationship at the type level

At the instance level, we are now able to model the peer-to-peer example, as well as our uncertainties and decisions, as we explained in section 4.2.2. Figures 6.8 - 6.12 show respectively the peer-to-peer example, its corresponding DRUIDE model, the mega-model that links both of them, the *Localization* traces, and the *Operationalization* traces in D-MMINT. In other terms, these figures represent D-MMINT implementation of the object diagram presented in Fig 4.2.



```
platform:/resource/MODELS20/Peer2Peer/Peer2Peer_duide.statemachine
  State Machine
    Initial State Idle
    State Seeding
    State Leeching
    State Finishing
    Transition start
    Transition share
    Transition restart
    Transition cancel
    Transition cancel
    Transition competed
    Transition completed
    Transition completed
    Transition peersDone
```

Fig. 6.8. A screen shot of the Peer-to-Peer example modeled in D-MMINT



```
platform:/resource/MODELS20/Peer2Peer/SMUncertainty.duide
  Druide Model
    DUncertainty I don't know how the seeding will be initiated,
    DUncertainty I don't know if users can resart downloads, or this should be done another way ?
    DUncertainty I need to decide what happens once a download is complete
    DDecision Can users initiate seeding ?
    DDecision Can users restart downloads ?
    DDecision What happens when a download is complete   ?
    DDecision Benevolent ?
    DDecision Selfish ?
    DDecision Compromise   ?
    DRephrasing Dependency
    DExcludes
    DExcludes
    DExcludes
    DExcludes
    DPolar
    DClosed Ended
    DDependency Formula A implies B ?
    DDependency Formula A implies not B
```

Fig. 6.9. A screen shot of the DRUIDE model concerning the peer-to-peer example modeled in D-MMINT

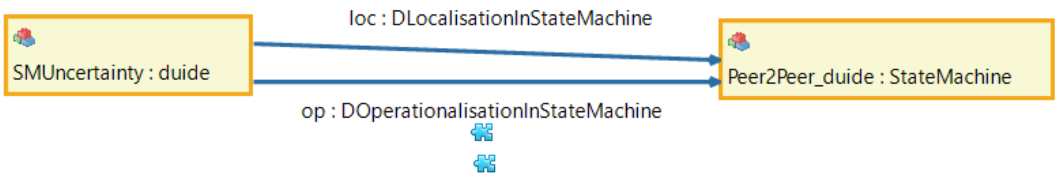

loc : DLocalisationInStateMachine
SMUncertainty : duide
Peer2Peer_duide : StateMachine
op : DOperationalisationInStateMachine

Fig. 6.10. A screen shot of the mega-model of the peer-to-peer example with the Druide model in D-Mmint

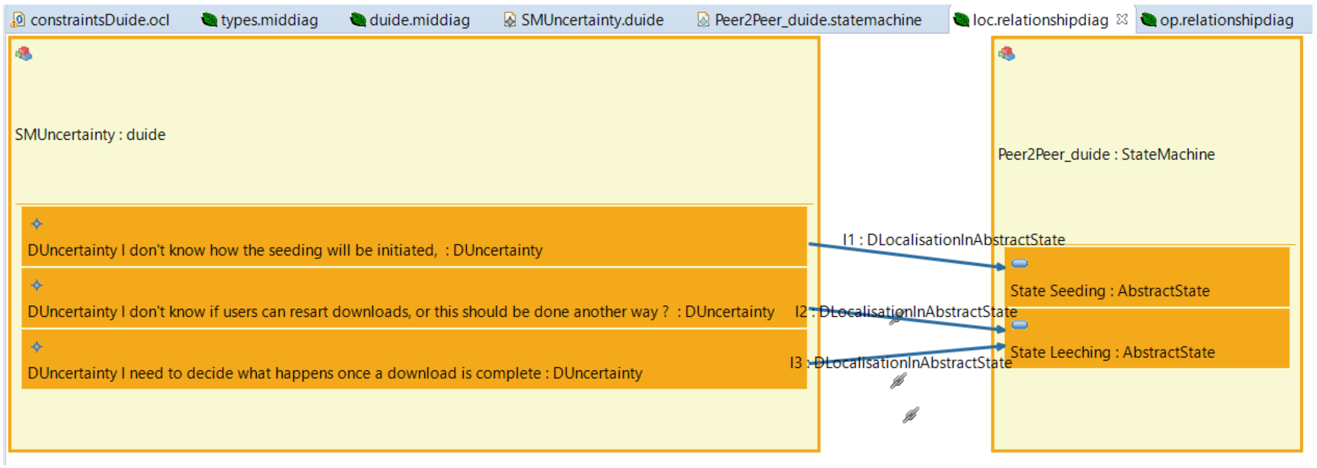

constraintsDuide.ocl
types.middiag
duide.middiag
SMUncertainty.duide
Peer2Peer_duide.statemachine
loc.relationshipdiag
op.relationshipdiag
SMUncertainty : duide
DUncertainty I don't know how the seeding will be initiated, : DUncertainty
DUncertainty I don't know if users can resart downloads, or this should be done another way ? : DUncertainty
DUncertainty I need to decide what happens once a download is complete : DUncertainty
Peer2Peer_duide : StateMachine
State Seeding : AbstractState
State Leeching : AbstractState
l1 : DLocalisationInAbstractState
l2 : DLocalisationInAbstractState
l3 : DLocalisationInAbstractState

Fig. 6.11. A screen shot of the *Localization* traces of the peer-to-peer example in D-Mmint

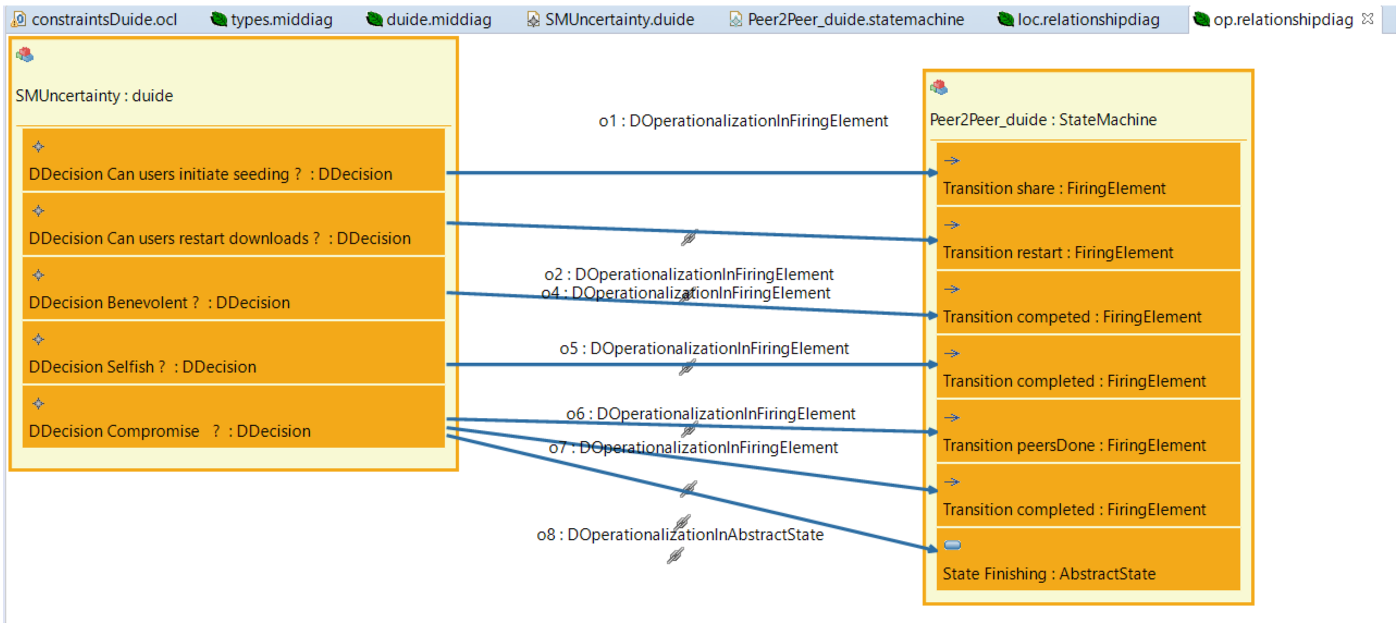
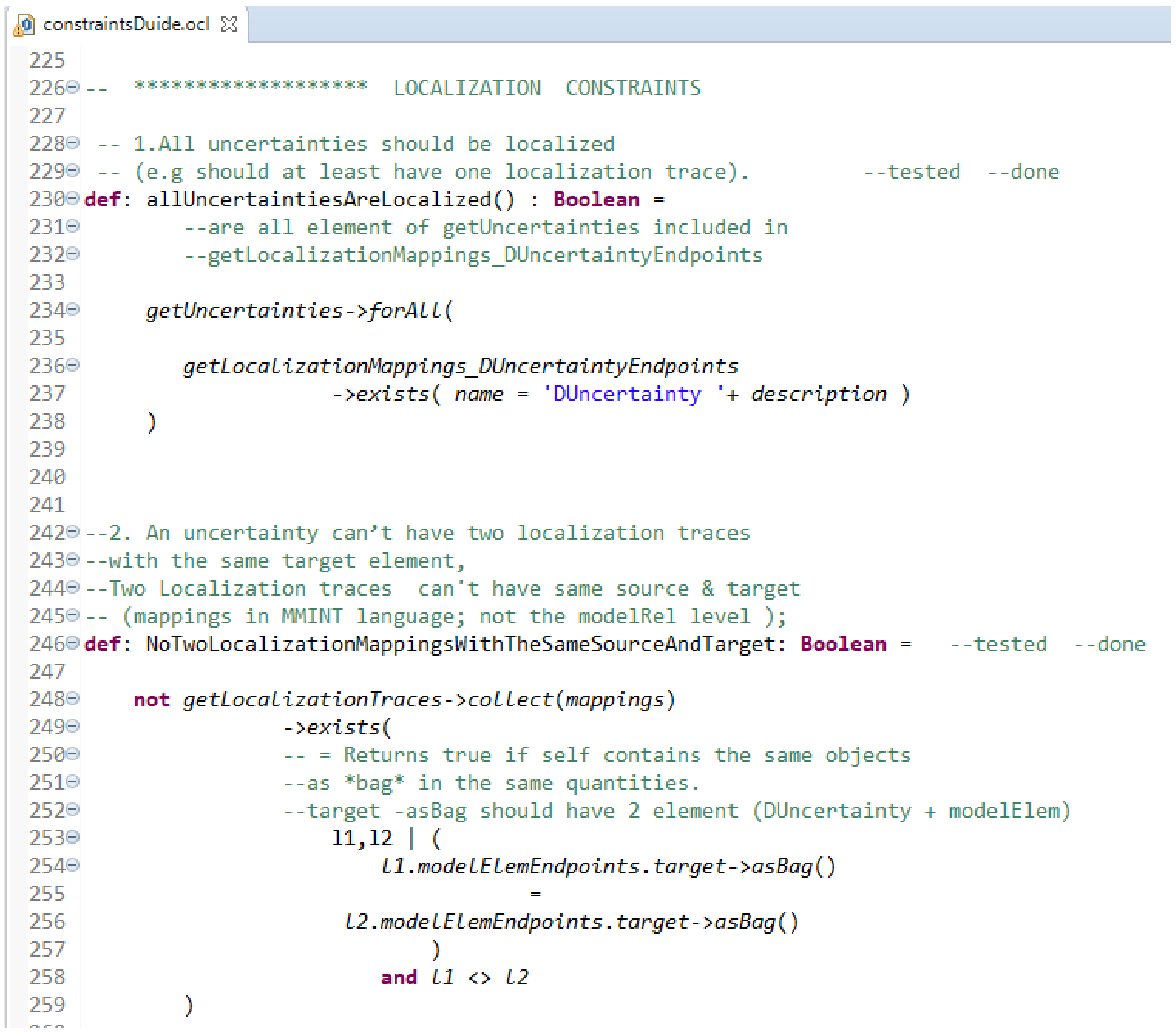

constraintsDuide.ocl
types.middiag
duide.middiag
SMUncertainty.duide
Peer2Peer_duide.statemachine
loc.relationshipdiag
op.relationshipdiag
SMUncertainty : duide
DDecision Can users initiate seeding ? : DDecision
DDecision Can users restart downloads ? : DDecision
DDecision Benevolent ? : DDecision
DDecision Selfish ? : DDecision
DDecision Compromise ? : DDecision
Peer2Peer_duide : StateMachine
Transition share : FiringElement
Transition restart : FiringElement
Transition competed : FiringElement
Transition completed : FiringElement
Transition peersDone : FiringElement
Transition completed : FiringElement
State Finishing : AbstractState
o1 : DOperationalizationInFiringElement
o2 : DOperationalizationInFiringElement
o4 : DOperationalizationInFiringElement
o5 : DOperationalizationInFiringElement
o6 : DOperationalizationInFiringElement
o7 : DOperationalizationInFiringElement
o8 : DOperationalizationInAbstractState

Fig. 6.12. A screen shot of the *Operationalization* traces of the peer-to-peer example in D-Mmint

Concerning the constraints implementation in MMINT, we have chosen Eclipse-OCL[3]. Eclipse-OCL can evaluate OCL constraints and queries on Ecore models. Besides, it is already supported in MMINT [7]. Fig 6.13 shows a fragment of the OCL file that implements the constraints. The full file with all the OCL constraints can be found in **Appendix A**. Figures 6.14 - 6.16 present the procedure of verifying an OCL constraint that holds over the peer-to-peer example in D-MMINT. Figures 6.18 - 6.19 present the result of checking an OCL constraint on a counter example model that breaks it.


constraintsDuide.ocl

```

--  *******************  LOCALIZATION  CONSTRAINTS

 -- 1.All uncertainties should be localized
 -- (e.g should at least have one localization trace).        --tested  --done
def: allUncertaintiesAreLocalized() : Boolean =
        --are all element of getUncertainties included in
        --getLocalizationMappings_DUncertaintyEndpoints

     getUncertainties->forAll(

        getLocalizationMappings_DUncertaintyEndpoints
                    ->exists( name = 'DUncertainty '+ description )
     )



--2. An uncertainty can’t have two localization traces
--with the same target element,
--Two Localization traces  can't have same source & target
-- (mappings in MMINT language; not the modelRel level );
def: NoTwoLocalizationMappingsWithTheSameSourceAndTarget: Boolean =   --tested  --done

    not getLocalizationTraces->collect(mappings)
                ->exists(
                -- = Returns true if self contains the same objects
                --as *bag* in the same quantities.
                --target -asBag should have 2 element (DUncertainty + modelElem)
                    l1,l2 | (
                        l1.modelElemEndpoints.target->asBag()
                                    =
                     l2.modelElemEndpoints.target->asBag()
                            )
                        and l1 <> l2
        )
```

Fig. 6.13. Some of the constraints implemented in OCL

Using D-MMINT, we were able to model all of the examples presented earlier in this thesis. Specifically, we were also able to implement the case study evaluation, as presented in Fig 6.21. Besides, D-MMINT allowed us to add the inter-model uncertainties on the mappings as sketched in Fig 5.7. Figures 6.22 - 6.25 show how this was done in D-MMINT.

[3]`https://projects.eclipse.org/projects/modeling.mdt.ocl`. URL accessed on September 13th, 2020.

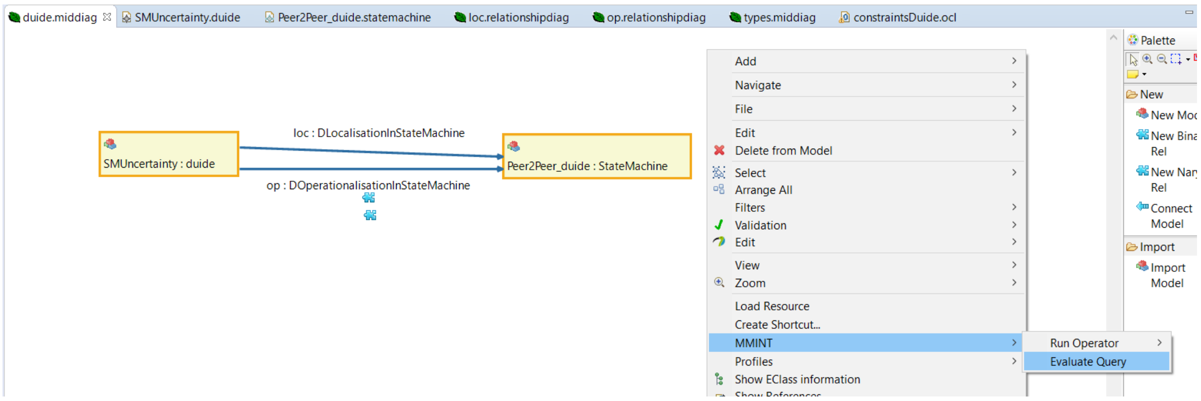


Fig. 6.14. Step 1 - Select the *Evaluate Query* option from the *MMINT* menu

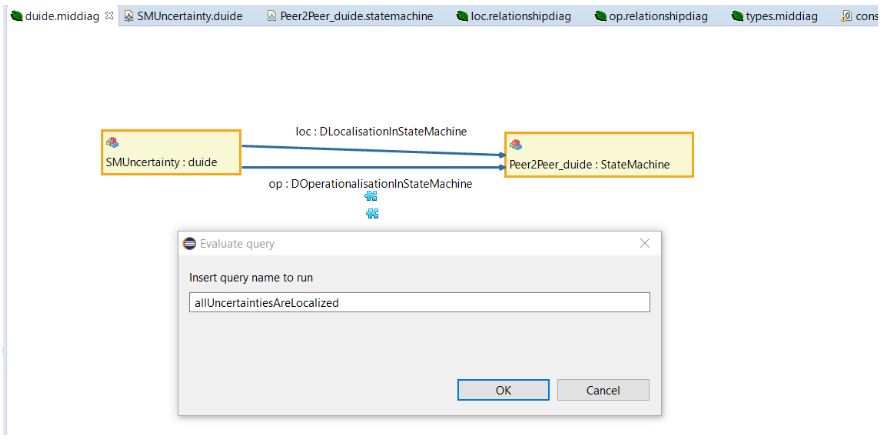


Fig. 6.15. Step 2 - Specify the constraint name and click *OK*

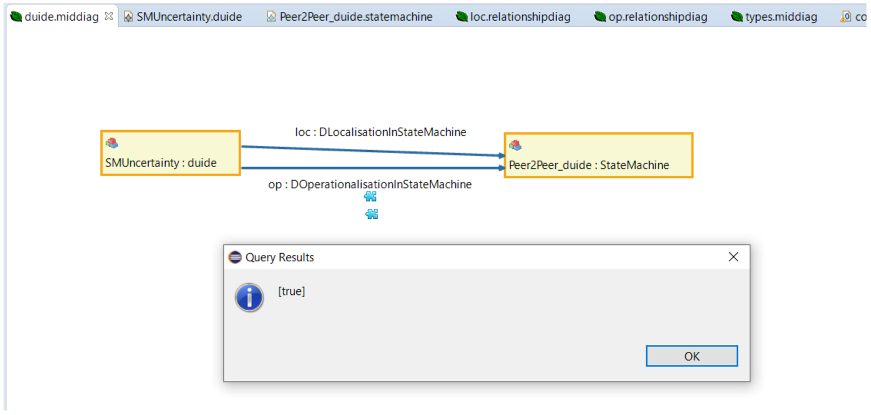
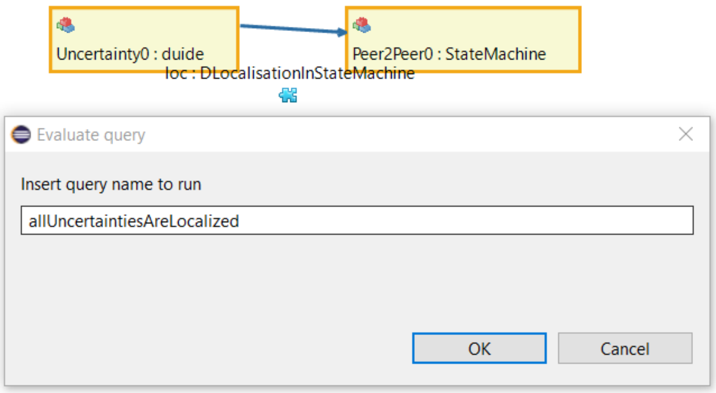


Fig. 6.16. Step 3 - Get the Evaluation result

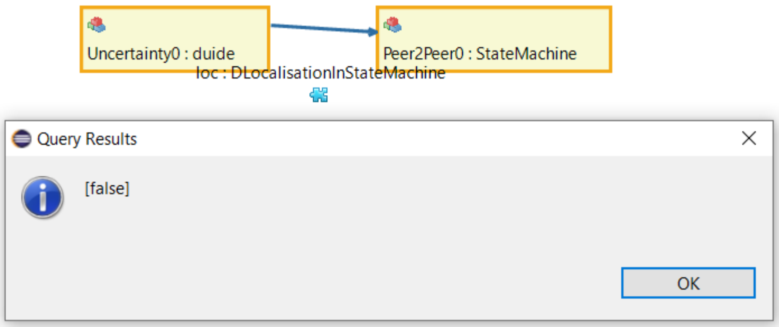
Fig. 6.17. The steps to verify an OCL constraint



Insert query name to run

allUncertaintiesAreLocalized

Fig. 6.18. Specify the constraint name and click *OK*



[false]

Fig. 6.19. Get the Evaluation result

Fig. 6.20. The result of checking an OCL constraint on a counter example peer-to-peer model that breaks it.

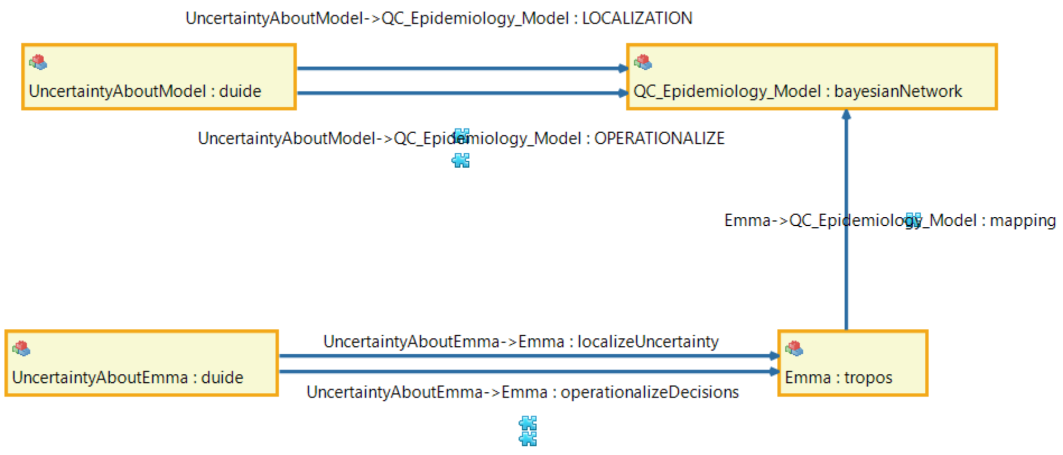

UncertaintyAboutModel->QC_Epidemiology_Model : LOCALIZATION
UncertaintyAboutModel : duide
QC_Epidemiology_Model : bayesianNetwork
UncertaintyAboutModel->QC_Epidemiology_Model : OPERATIONALIZE
Emma->QC_Epidemiology_Model : mapping
UncertaintyAboutEmma->Emma : localizeUncertainty
UncertaintyAboutEmma : duide
Emma : tropos
UncertaintyAboutEmma->Emma : operationalizeDecisions

Fig. 6.21. Implementation of the case study in D-Mmint

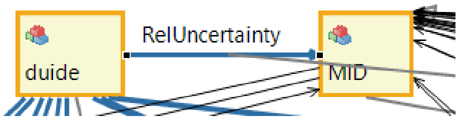


Fig. 6.22. At the type level, creating a *RelUncertainty* relationship between DRUIDE model and the *MID* model

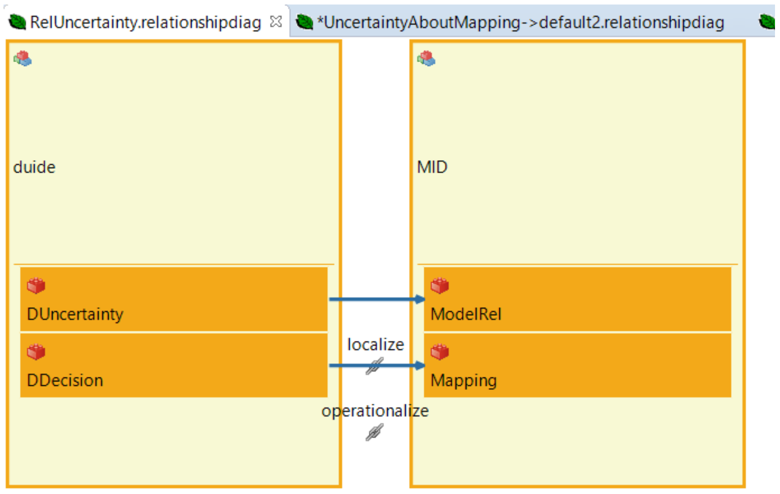

RelUncertainty.relationshipdiag
*UncertaintyAboutMapping->default2.relationshipdiag
duide
DUncertainty
DDecision
localize
operationalize
MID
ModelRel
Mapping

Fig. 6.23. Definition of the *RelUncertainty* at the type level

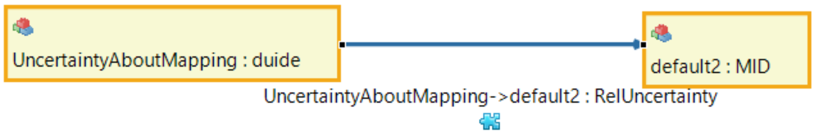

UncertaintyAboutMapping : duide
default2 : MID
UncertaintyAboutMapping->default2 : RelUncertainty

Fig. 6.24. At the instance level, creating an instance of the *RelUncertainty* between the DRUIDE model of the mapping uncertainties and the *MID* of the case study (Fig 6.21)

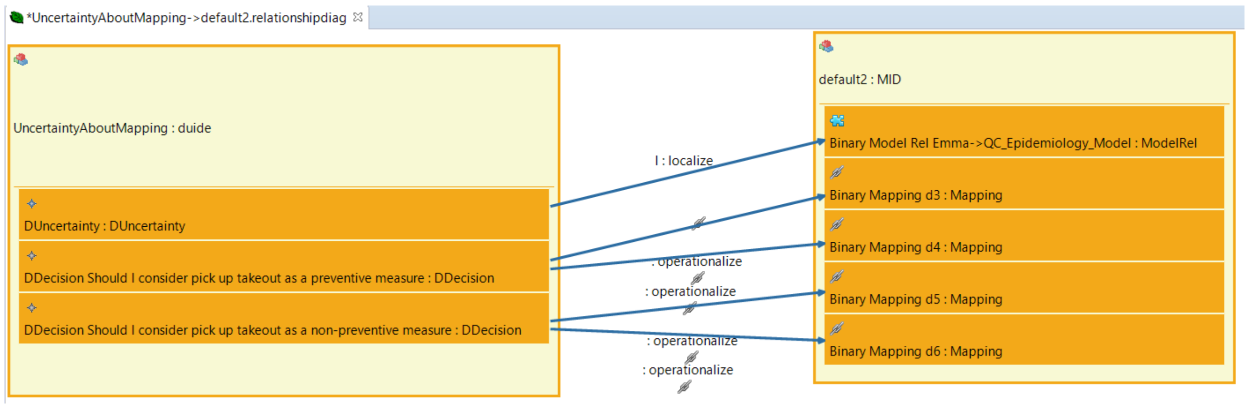

*UncertaintyAboutMapping->default2.relationshipdiag
UncertaintyAboutMapping : duide
DUncertainty : DUncertainty
DDecision Should I consider pick up takeout as a preventive measure : DDecision
DDecision Should I consider pick up takeout as a non-preventive measure : DDecision
l : localize
: operationalize
: operationalize
: operationalize
: operationalize
default2 : MID
Binary Model Rel Emma->QC_Epidemiology_Model : ModelRel
Binary Mapping d3 : Mapping
Binary Mapping d4 : Mapping
Binary Mapping d5 : Mapping
Binary Mapping d6 : Mapping

Fig. 6.25. At the instance level, localizing the mapping uncertainty and operationalizing the mapping decisions

Fig. 6.26. The inter-model mapping uncertainties implemented in D-MMINT

Although D-Mmint has proven to be more adequate than our initially proposed ATOMPM-based workflow (section 6.1), it still imposed some usage restrictions.

First, the *Operationalization* and *Localization* relationships are not part of the Druide Eclipse plugin. This means that the user should define these two relationships at the type level, every time he wants to express some design-time uncertainty on a model. This entails that the user has previous knowledge of Druide concepts and methodology, and that he will respect them. For instance, the user will respect that uncertainties are localized, while decisions are operationalized, and not the other way around, or, he will not introduce other types of traces. On the other hand, this is beneficial because the user has explicit control on specifying which model elements can the uncertainty be localized in, and which model elements can be the result of an operationalization operation.

Second, the concept of *mappings* in Mmint is used to create model transformations[4]. In our case, we are considering the Mmint mappings as simple traces, without any semantics.

Finally, the cross-language OCL constraints we implemented are not generic. In fact, they are written specifically for the State Machines meta-model. In order to make them generic and applicable to any model, we would have to either, alter the Druide Ecore model and make the *DUncertainty* and *DDecision* elements incorporate references to the objects that respectively localize or operationalize them, which means giving up the traces objects and stepping out of the mega-modelling mindset; or writing the constraints at a higher level of abstraction, which will significantly increase the complexity of their implementation.

In this chapter, we presented the current state of our investigation to provide tool support for our work. In the next chapter, we discuss Druide model and approach by highlighting its benefits as well as its limitations.

[4]Examples of such transformations, and of the correct usage of the mappings can be found here: `https://www.youtube.com/watch?v=zXFKuP7qYpA`. URL accessed on September 13th, 2020.

Chapter 7

Discussion

In this chapter, we discuss our work. We start by stating the benefits of DRUIDE, then we expose its limitations. We also mention some potential ideas for future work.

7.1. Benefits

In order to highlight the benefits of our work, we start by showing its relevance, then, we compare it to existing work on modelling uncertainty. Afterwards, we highlight its usefulness regarding the DETUM framework (section 2.3.3). Finally, we mention the benefits of having tool support for the articulation of uncertainty.

7.1.1. Relevance of our work

Uncertainty is intrinsic in the software modelling process. Thus, uncertainty-aware modelling is more truthful and more realistic than the usual modelling. That's why, uncertainty modelling has been getting a lot of attention in the recent years [**44, 15, 12, 32**].

Besides, given the current situation and the uncertainty caused by the onset of the COVID-19 pandemic, we believe that the need for a language and a tool for modelling in the presence of uncertainty has become even more important and urgent.

As part of our future work, we are planning on empirically validating the need for uncertainty-aware modelling approaches by conducting a large scale survey.

7.1.2. Comparaison with the state of the art

In order to highlight better the benefits of our work, we compare it to existing work on modelling uncertainty.

In [**17**], Famelis et al. proposed MAV-Vis, a notation for partial models based on MAVO (sections 2.2 and 2.3.2). Similar to our work, MAV-Vis can always be used with abstract syntax, since Class Diagrams are used to express any model in its abstract syntax. Different from our work, MAV-Vis cannot annotate arbitrary concrete syntaxes. Finally, MAV-Vis does not address the OW partiality, that is expressed at the model level. On the other side, DRUIDE with its combination with the D-MMINT tooling, is based on mega-modelling. Thus, it supports multiple abstraction layers, and can express intra-model and inter-model partiality.

In [**41**], the authors suggest that uncertainty is a three dimensional concept defined by its nature, its level and its location and propose a matrix to characterize it (sections 2.2 and 2.2.1). The matrix they propose constitutes a snapshot of the uncertainties in a system at a particular point in time. So, they do not provide support to express the articulation of the evolution of uncertainty. Moreover, their tool requires the use of a separate artifact, besides the model.

In [**44**], Zhang et al. propose U-Model: a conceptual model for uncertainty specifically designed for Cyber-Physical Systems (CPSs) (section 2.2.2). Similar to our work, they propose a model for uncertainty, different from our work, they keep it separate from the CPS model. In another work [**46**], the same authors propose the U-RUCM methodology and tool to identify and specify uncertainty as part of system requirements. Since their work only considers uncertainty in the scope of use-case modelling, it differs from the generic approach we propose in this thesis.

In [**47**], the authors propose a mathematically sound technique for modelling uncertainty based on Bayesian belief networks (section 5.2.2), and they demonstrate its applicability to model uncertainty in the context of a software engineering situation. The main limitation of this work is that it is based on probabilities, and thus its application use cases are very limited. Besides, this probabilities-based technique fails to express the modellers thoughts or represent their doubts.

To conclude, considering the limitations of the above works, we argue that we are proposing a more complete and more expressive uncertainty-aware modelling language than the ones proposed in the literature.

7.1.3. Adding support to the articulation stage of the DETUM model

In this section, we present the usefulness of our work by grounding it on existing work on modelling uncertainty. Specifically, DRUIDE can be seen as a way of adding support for the evolution of the articulation stage of the uncertainty in the DETUM framework (section 2.3.3). We will explain this below.

The main limitation of the DETUM model and of partial models in general (section 2.3), is that the authors introduce and tackle design-time uncertainty as a pre-defined existing set of alternative possible designs. The authors assume that the developers have already identified the design uncertainties, articulated the corresponding candidate solutions, elicited how each one is implemented, and somehow deduced the partial model that compactly yet precisely encodes the entire set of all alternative possible designs [**15**].

Specifically, the DeTUM model starts-off with a finite set of design alternatives, that are merged into one partial model with several *Maybe*-annotated elements and a *May* formula that captures the allowable configurations of the *Maybe* elements. Thus, partial models only allow modellers to distinguish between the *Maybe* elements and the *True* elements, and to consider different design possibilities. They do not answer questions like: *Why is that element Maybe-annotated?*, *How did we come to annotate that element as Maybe?*.

On the other hand, the work we propose in this thesis fills this gap as it provides a language and an approach to explicitly express, evolve and elicit the candidate solutions. Specifically, the result of the *Operationalization* operation is Druide annotates the system elements as *Maybe*. Thus, Druide models allow modellers to answer the above questions. Fig 7.1 schematically shows Druide impact (orange inclined line) on the DeTUM articulation stage.

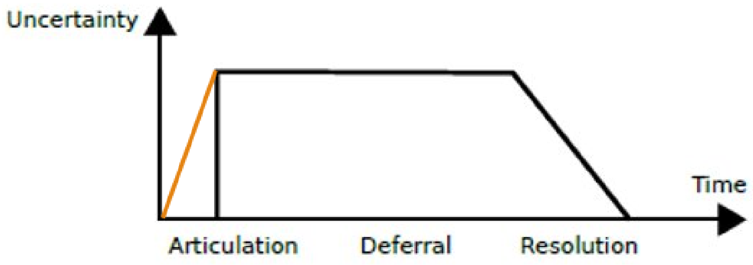


Fig. 7.1. Druide as a support for the evolution of the articulation stage of the DeTUM model

The unrealistic assumption of the DeTUM framework (the assumption that modellers would start-off with a finite set of design alternatives) have hindered its applicability in real life. We envision that semantically grounding Druide on DeTUM can make the DeTUM framework applicable and accessible in real-life scenarios. We will detail our vision to do this later in Chapter 8.

7.1.4. Tool support benefits

In this section, we present the benefits of having tool support for uncertainty-aware modelling.

Leveraging an uncertainty-aware modelling tool, modellers would no longer have to stop working, switch modelling environments, or maintain a separate log of their uncertainty and decisions. Instead, they would be able to perform their modelling tasks while expressing and localizing their uncertainty, and modelling and operationalizing the corresponding decisions, all within the same modelling environment. Intuitively, this would have a positive impact on their productivity.

As part of our future work, we are planning on empirically validating the impact of using an uncertainty-aware modelling tool on the performance of the modellers.

7.2. Limitations

In this section, we enumerate and discuss the limitations of our work and we give pointers for future improvements.

First, we have only considered the *May* partiality type in the context of this thesis. In the future, we envision considering other partiality types.

Second, we believe that the model-related constraints we presented in Chapter 3 are not enough. The constraints presented in Table 3.1 are all dependency related constraints. We believe that we need to consider other constraints, for instance, in relation with other partiality types, or regarding the allowed compositions between the uncertainty levels and natures.

Third, we find the decomposition approach regarding the evolution of the decisions that we propose in this work; e.g, from *DOpenEnded* to *DClosedEnded* to *Boolean* decisions, rather simplistic. In the future, we plan to investigate other decomposition approaches. Moreover, we want to investigate the potential relationship between the evolution of the decisions types during this decomposition, and the possible evolution of the corresponding uncertainty level.

Besides, the logical dependency formula is currently typed as a simple *String* (Fig 3.1). As an optimization, we are looking into using more sophisticated representations that are better suited for propositional logic.

Finally, as we deduced after evaluating DRUIDE on the UMLetBug example (section 4.2.3), DRUIDE can complicate simple scenarios that concern small model variances. Thus, as part of our future work, we aim to clearly frame DRUIDE's most suitable use cases.

In this chapter, we highlighted DRUIDE benefits and exposed its limitations. In the next chapter, we conclude this thesis and present our work in progress concerning our near-future research directions.

Chapter 8

Conclusion and Next steps

In this chapter, we conclude this thesis, then we outline our future work.

8.1. Summary

In this thesis, we presented our proposal, DRUIDE, an uncertainty-aware modelling language and approach, and illustrated it on a worked example. Besides, we evaluated our work not only on two worked software engineering scenarios, but also on a real case study based on the COVID-19 pandemic. We also conducted a post-study questionnaire with the three researchers who participated in the case study. In order to prove the feasibility of our approach, we provided two tool supports and discussed them. Finally, we highlighted both the benefits and the limitations of our work.

8.2. Next steps

In this section, we present our next steps.

8.2.1. DRUIDE Semantics

The next step in this project consists of introducing DRUIDE semantics. In fact, we want to create a semantic mapping of DRUIDE operators that is grounded on existing work on representing uncertainty; partial models (presented in section 2.3). We note that for this part, we are also only considering *MAY* partiality. Creating this kind of semantic mapping requires three steps: (a) studying the semantics of partial models, (b) defining the appropriate DRUIDE operators, and (c) introducing a semantic mapping between them. We describe these steps below, and focus on the resolving uncertainty operators to give an example.

8.2.1.1. **Partial models formal specification**. We have been finalizing the formal specification of design-time uncertainty with partial models, over the DETUM life cycle (section

2.3.3). In the rest of this section, we briefly report the refinement operators for working with partial models. Specifically, we focus on the manual refinement that involves manually changing the annotations of the partial models elements and the *May* formula to decrease the level of uncertainty.

In partial models, 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. Conversely, invoking *Drop* for a partial model element amounts to deciding that it should not be part of the model.

8.2.1.2. DRUIDE ***Concretization*** **operators**. We are investigating the introduction of *Concretization* operators in DRUIDE. The *Concretization* operators will be used to resolve uncertainty by making the decision of existence or not of the *Maybe*-annotated system elements that operationalize some *DPolar* decision. Our initial plan is to distinguish between two types of *Concretization* operators:

- ***Adopt*** : Adopting a *DPolar* design-decision means the modellers decided to make that decision. Thus, all the *Maybe*-annotated elements that elicit it should become *True* elements.
- ***Discard***: Discarding a *DPolar* design-decision means the modellers decided not to opt for that decision in the system's design. Therefore, all the *Maybe*-annotated elements that elicit it should be removed from the model.

8.2.1.3. **The semantic mapping**. Roughly, we intend to semantically map the *Adopt* and the *Discard* operators respectively with the *Keep* and *Drop* operators. In other words, we plan to define DRUIDE *Concretization* operators semantics in terms of partial models refinement operators semantics, and evaluate the correctness of this mapping.

Finally, we note that the above refinement operators are just provided for illustration purposes, and are not the only operators we are planning to map. In fact, we are also investigating that the semantic mapping from DRUIDE to partial models would provide automated support for the DETUM high level uncertainty articulation operators. Specifically, we are considering representing each element of the model under construction by a logical preposition. Then, transforming the *DPolar* Boolean decisions that have been operationalized by the modellers, into logical prepositions that represent a conjunction of all the prepositions of the system elements that operationalize them. Since an uncertainty can have several *DPolar* design decisions that are related using only logical dependencies (The rationale behind the 5th constraint in Table 3.1), we estimate that we can generate an uncertainty-level formula that encodes all the different designs and the dependencies between them. Afterwards, we can compact all these formulas into a model-level prepositional formula. We envision this as a process of automating the generation of partial models with their *May* formulas. This, in turn, will allow modellers to take advantage of approaches that leverage partial models for reasoning [**12**], refinement [**33**], and transformations [**16**].

8.2.2. Automation-supported modelling

Our future work includes proposing an automation-supported uncertainty modelling approach. In fact, we aim to provide a framework that mines artifacts containing developers and stakeholders interactions, and automatically generates DRUIDE models and traces them to the system models.

As a first step in this research direction, we started by investigating the Mining Software Repositories (MSR) field. This field analyzes and cross-links the collaborative repositories data to uncover interesting and actionable information about software systems [**21**]. As part of our early investigation, we elaborated a literature review of several software engineering tasks that made use of the MSR activities. In the rest of this section, we report some potential kick-off ideas, structured by software engineering application domain.

- **Emotion mining** relates to the identification of the presence of human emotions from artifacts produced by humans.
  We plan to introduce the *confusion* emotion, and mine it as a potential indicator for the presence of design uncertainty.
- **Concept location** maps concepts expressed in natural language by the programmers to the relevant parts of the source code.
  We envision proposing a similar approach to automatically localize the uncertainty in the corresponding parts of the model.
- **Requirements clarification** refers to the process in which requirements evolve from initial ideas to the implementation of a stable requirement.
  We plan to propose a similar approach to trace and gradually model the evolution of the design decisions from high-level OpenEndend questions to implementable DPolar design decisions.
- **Requirements tracing** is used to verify that all the requirements have been implemented at the end. The main idea is to map elements of high-level artifacts such as requirements, to elements of low-level artifacts such as design.
  We want to investigate the possibility to use such mappings to automatically create operationalization traces that link DPolar design decisions with their corresponding elicitations in the system elements.
- **Argument mining** is the task of identifying argumentative contents and components in natural language texts.

  Our initial idea is to use the argumentation identification to detect persuasion between modellers, and check if the result of that persuasion is the resolving of some design uncertainty.
- **Mining software design information** aims to capture and communicate the latent and distributed design information.
  Intuitively, we think that finding and extracting design information from discussions with stakeholders can help resolve the modellers uncertainties [**40**].

Mining developers interactions for uncertainty modelling will be the focus of my PhD research.

References

[1] Atlantic meta-model zoo. `https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=ZooFederation`. URL accessed on September 13th, 2020.

[2] Philip A Bernstein : Applying model management to classical meta data problems. *In CIDR*, volume 2003, pages 209–220. Citeseer, 2003.

[3] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia et John Mylopoulos : Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[4] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert B France, Jean-Marc Jézéquel et Jeff Gray : Globalizing modeling languages. *Computer*, 47(6):68–71, 2014.

[5] Deepak Dhungana, Paul Grünbacher et Rick Rabiser : Domain-specific adaptations of product line variability modeling. *In Working Conference on Method Engineering*, pages 238–251. Springer, 2007.

[6] Deepak Dhungana, Paul Grünbacher et Rick Rabiser : The dopler meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011.

[7] Alessio Di Sandro, Sahar Kokaly, Rick Salay et Marsha Chechik : Querying automotive system models and safety artifacts with mmint and viatra. *In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 2–11. IEEE, 2019.

[8] Alessio Di Sandro, Rick Salay, Michalis Famelis, Sahar Kokaly et Marsha Chechik : Mmint: A graphical tool for interactive model management. *In P&D@ MoDELS*, pages 16–19, 2015.

[9] Matthew Emerson et Janos Sztipanovits : Techniques for metamodel composition. *In OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[10] Romina Eramo, Alfonso Pierantonio et Gianni Rosa : Managing Uncertainty in Bidirectional Model Transformations. *In Proc. of SLE'15*, pages 49–58. ACM, 2015.

[11] Wolfgang Ertel : Reasoning with uncertainty. *In Introduction to Artificial Intelligence*, pages 125–174. Springer, 2017.

[12] Michais Famelis, Rick Salay et Marsha Chechik : Partial models: Towards modeling and reasoning with uncertainty. *In 2012 34th International Conference on Software Engineering (ICSE)*, pages 573–583. IEEE, 2012.

[13] Michalis Famelis, Naama Ben-David, Alessio Di Sandro, Rick Salay et Marsha Chechik : Mu-mmint: an ide for model uncertainty. *In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 697–700. IEEE, 2015.

[14] Michalis Famelis, Shoham Ben-David, Marsha Chechik et Rick Salay : Partial models: A position paper. *In Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, page 1. ACM, 2011.

[15] Michalis Famelis et Marsha Chechik : Managing design-time uncertainty. *Software & Systems Modeling*, 18(2):1249–1284, 2019.

[16] Michalis Famelis, Rick Salay, Alessio Di Sandro et Marsha Chechik : Transformation of models containing uncertainty. *In International Conference on Model Driven Engineering Languages and Systems*, pages 673–689. Springer, 2013.

[17] Michalis Famelis et Stephanie Santosa : Mav-vis: a notation for model uncertainty. *In 2013 5th International Workshop on Modeling in Software Engineering (MiSE)*, pages 7–12. IEEE, 2013.

[18] T. Fukamachi, N. Ubayashi, S. Hosoai et Y. Kamei : Modularity for Uncertainty. *In Proc. of MODELS'15*, pages 7–12, May 2015.

[19] Miguel Gamboa et Eugene Syriani : Improving user productivity in modeling tools by explicitly modeling workflows. *Software & Systems Modeling*, 18(4):2441–2463, 2019.

[20] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel et Andreas Wortmann : Composition of heterogeneous modeling languages. *In International Conference on Model-Driven Engineering and Software Development*, pages 45–66. Springer, 2015.

[21] Ahmed E Hassan : The road ahead for mining software repositories. *In 2008 Frontiers of Software Maintenance*, pages 48–57. IEEE, 2008.

[22] Regina Hebig, Andreas Seibel et Holger Giese : On the unification of megamodels. *Electronic Communications of the EASST*, 42, 2012.

[23] Kurt Jensen et Lars M Kristensen : *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.

[24] Holger Krahn, Bernhard Rumpe et Steven Völkel : Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.

[25] Alexei Lapouchnian et John Mylopoulos : Modeling domain variability in requirements engineering with contexts. *In International Conference on Conceptual Modeling*, pages 115–130. Springer, 2009.

[26] Eva Millán, Tomasz Loboda et Jose Luis Pérez-De-La-Cruz : Bayesian networks for student model engineering. *Computers & Education*, 55(4):1663–1683, 2010.

[27] OMG : Decision model and notation, version 1.3, March 2020.

[28] OCL OMG : 2.0 specification. *Object Management Group, Final Adopted Specification*, 2005.

[29] Klaus Pohl, Günter Böckle et Frank J van Der Linden : *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[30] Rick Salay et Marsha Chechik : A generalized formal framework for partial modeling. *In International Conference on Fundamental Approaches to Software Engineering*, pages 133–148. Springer, 2015.

[31] Rick Salay, Marsha Chechik, Steve Easterbrook, Zinovy Diskin, Pete McCormick, Shiva Nejati, Mehrdad Sabetzadeh et Petcharat Viriyakattiyaporn : An eclipse-based tool framework for software model management. *In Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 55–59, 2007.

[32] Rick Salay, Marsha Chechik, Jennifer Horkoff et Alessio Di Sandro : Managing requirements uncertainty with partial models. *Requirements Engineering*, 18(2):107–128, 2013.

[33] Rick Salay, Michalis Famelis et Marsha Chechik : Language independent refinement using partial modeling. *In International Conference on Fundamental Approaches to Software Engineering*, pages 224–239. Springer, 2012.

[34] Rick SALAY, Sahar KOKALY, Alessio DI SANDRO et Marsha CHECHIK : Enriching megamodel management with collection-based operators. *In 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 236–245. IEEE, 2015.

[35] Klaus SCHMID, Rick RABISER et Paul GRÜNBACHER : A comparison of decision modeling approaches in product lines. *In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126, 2011.

[36] Oszkár SEMERÁTH et Dániel VARRÓ : Graph constraint evaluation over partial models by constraint rewriting. *In* Esther GUERRA et Mark van den BRAND, éditeurs : *Theory and Practice of Model Transformation*, pages 138–154, Cham, 2017. Springer International Publishing.

[37] Eugene SYRIANI, Hans VANGHELUWE, Raphael MANNADIAR, Conner HANSEN, Simon VAN MIERLO et Huseyin ERGIN : Atompm: A web-based modeling environment. *In Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pages 21–25, 2013.

[38] Federico TOMASSETTI, Antonio VETRÓ, Marco TORCHIANO, Markus VOELTER et Bernd KOLB : A model-based approach to language integration. *In 2013 5th International Workshop on Modeling in Software Engineering (MiSE)*, pages 76–81. IEEE, 2013.

[39] Axel VAN LAMSWEERDE : *Requirements engineering: From system goals to UML models to software*, volume 10. Chichester, UK: John Wiley & Sons, 2009.

[40] Giovanni VIVIANI, Michalis FAMELIS, Xin XIA, Calahan JANIK-JONES et Gail C MURPHY : Locating latent design information in developer discussions: A study on pull requests. *IEEE Transactions on Software Engineering*, 2019.

[41] Warren E WALKER, Poul HARREMOËS, Jan ROTMANS, Jeroen P VAN DER SLUIJS, Marjolein BA VAN ASSELT, Peter JANSSEN et Martin P Krayer von KRAUSS : Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support. *Integrated assessment*, 4(1):5–17, 2003.

[42] K. WATANABE, N. UBAYASHI, T. FUKAMACHI, S. NAKAMURA, H. MURAOKA et Y. KAMEI : iarch-u: Interface-centric integrated uncertainty-aware development environment. *In Proc. of MiSE'17*, May 2017.

[43] Tao YUE, Lionel C BRIAND et Yvan LABICHE : Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):5, 2013.

[44] Man ZHANG, Bran SELIC, Shaukat ALI, Tao YUE, Oscar OKARIZ et Roland NORGREN : Understanding uncertainty in cyber-physical systems: a conceptual model. *In European conference on modelling foundations and applications*, pages 247–264. Springer, 2016.

[45] Man ZHANG, Bran SELIC, Shaukat ALI, Tao YUE, Oscar OKARIZ et Roland NORGREN : Understanding uncertainty in cyber-physical systems: a conceptual model. *In European conference on modelling foundations and applications*, pages 247–264. Springer, 2016.

[46] Man ZHANG, Tao YUE, Shaukat ALI, Bran SELIC, Oscar OKARIZ, Roland NORGRE et Karmele INTXAUSTI : Specifying uncertainty in use case models. *Journal of Systems and Software*, 144:573–603, 2018.

[47] Hadar ZIV, Debra RICHARDSON et René KLÖSCH : The uncertainty principle in software engineering. *In submitted to Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, 1997.

Appendix A

OCL constraints

```
import 'http://se.cs.toronto.edu/mmint/MID'
import 'http://se.cs.toronto.edu/modelepedia/StateMachine'
import 'http://geodes.iro.umontreal.ca/duide'


context mid::MID


----Some examples:

-------ToKnow OCL TYPE of something
 --directLocalizedUncertainDecisions->oclType()

-- return first ModelElement OCL Type
--def: e : ModelElement =
  ---ModelElement.allInstances()->asOrderedSet()->first()->oclType()


-- -- -- -- -- -- -- Helper OCL Queries:


                                ------------MMINT Helper Queries

-- return all Model Instances (in our case, Uncertainty & Peer2Peer &
operationalization & localization)
def: getAllModels : Collection(Model) =
            Model.allInstances()


-- return all BinaryModelRel
def: getAllBinaryModelRel :Collection (relationship::BinaryModelRel)=
                                Model.allInstances()
                                 ->select(oclIsTypeOf(relationship::BinaryModelRel))
                                  ->collect(oclAsType(relationship::BinaryModelRel))
                                  ->asSet()


-- return all Connected ModelElements Instances ( not clear, an extra DUncertainty )
def: getConnectedModelElements : Collection(ModelElement) =
  ModelElement.allInstances()


                                  ------------StateMachine Helper Queries

-- return all ModelElements Instances of type StateMachine
def: getStateMachines :Collection(statemachine::StateMachine) = Model.allInstances()
                                            ->collect(EMFInstanceRoot)

                                         ->select(oclIsTypeOf(statemachine::StateMachine)
                                         )
                                        ->collect(oclAsType(statemachine::StateMachine))


----return all states
def: getStates : Collection (statemachine::State) =
  let stateMachines = Model.allInstances()
                           ->collect(EMFInstanceRoot)
                           ->select(oclIsTypeOf(statemachine::StateMachine))
                           ->collect(oclAsType(statemachine::StateMachine)) in
  stateMachines->collect(states)


----return all transitions
def: getTransitions : Collection(statemachine::Transition) =
  let stateMachines = Model.allInstances()
                           ->collect(EMFInstanceRoot)
                           ->select(oclIsTypeOf(statemachine::StateMachine))
```

```
                            ->collect(oclAsType(statemachine::StateMachine)) in
    stateMachines->collect(transitions)
                 ->asSet()


                                        ------------Duide Helper Queries

  -- return all ModelElements Instances of type
  DuideModel
  def: getDuideModels :Collection(Model) = Model.allInstances()
                                                ->collect(EMFInstanceRoot)
                                             ->select(oclIsTypeOf(duide::DruideModel))
                                            ->collect(oclAsType(duide::DruideModel))



   -- return all dDecisions Instances --
   def: getDecisions :Collection(duide::DDecision) =

              let duideModels =            Model.allInstances()
                                             ->collect(EMFInstanceRoot)
                                             ->select(oclIsTypeOf(duide::DruideModel))
                                            ->collect(oclAsType(duide::DruideModel))   in

              duideModels->collect(dDecisions)



   -- return all Uncertainty Instances --
  def: getUncertainties :Collection(duide::DUncertainty) =

              let duideModels =            Model.allInstances()
                                             ->collect(EMFInstanceRoot)
                                             ->select(oclIsTypeOf(duide::DruideModel))
                                            ->collect(oclAsType(duide::DruideModel))   in

             duideModels->collect(dUncertainties)
                  --   ->select(e |
                  e.description.toLower().matches('.*know.*'))->oclType() --works



   -- return all dDependency Instances --
   def: getAllDependencies :Collection(duide::DDependency) =

              let duideModels =            Model.allInstances()
                                             ->collect(EMFInstanceRoot)
                                             ->select(oclIsTypeOf(duide::DruideModel))
                                            ->collect(oclAsType(duide::DruideModel))   in

              duideModels->collect(dDependencies)



  -- return DRephrasing dependencies
  def: getDRephrasingDependencies: Collection(duide::DRephrasingDependency) = -- TODO:
  change to Collection(duide::DDependency) + test where used
      getAllDependencies->select(oclIsTypeOf(duide::DRephrasingDependency))



  -- return DLogicalDependencies dependencies
  def: getDLogicalDependencies: Collection(duide::DLogicalDependency) =
     -- OclIsKindOf() Returns true if the type of self corresponds to the type or
     supertype of typespec, false otherwise.
      getAllDependencies->select(oclIsKindOf(duide::DLogicalDependency))




                                      ------------Traces Helper Queries

    -- return LocalizationTraces
```

```
 def: getLocalizationTraces :Collection (relationship::BinaryModelRel)=
           getAllBinaryModelRel ->select(e |
           e.metatypeUri.matches('http://se.cs.toronto.edu/mmint/DLocalisationInStateMachine'))


    -- return LocalizationTraces mappings
def: getLocalizationMappings:Collection(ModelElement) =
                  getLocalizationTraces->collect(mappings)


     -- return OperationalizationTraces
 def: getOperationalizationTraces :Collection (relationship::BinaryModelRel)=
           getAllBinaryModelRel ->select(e |
           e.metatypeUri.matches('http://se.cs.toronto.edu/mmint/DOperationalisationInStateMachine'))


    -- return OperationalizationTraces mappings
def: getOperationalizationMappings:Collection(ModelElement) =
                  getOperationalizationTraces->collect(mappings)


    -- return LocalizationTraces mappings  Endpoints   that are of type
    DUncertainty
def: getLocalizationMappings_DUncertaintyEndpoints:Collection(ModelEndpoint) =
         getLocalizationTraces->collect(mappings)
                              ->collect(modelElemEndpoints)
                              ->collect(target.oclAsType(ModelElement))
                              ->select(e|
                              e.metatypeUri.matches('http://geodes.iro.umontreal.ca/duide#//DUncertainty'))
                         --      ->select(e | e.name.toLower().matches('.*users.*'))
                         --works


    -- return LocalizationTraces mappings  Endpoints   that are of type
    DUncertainty
def: getOperationalizationMappings_DDecisionEndpoints:Collection(ModelElement) =
         getOperationalizationTraces->collect(mappings)
                              ->collect(modelElemEndpoints)
                              ->collect(target.oclAsType(ModelElement))
                              ->select(e|
                              e.metatypeUri.matches('http://geodes.iro.umontreal.ca/duide#//DDecision'))
                             -- ->collect(getEMFTypeObject()) -- dont work
                         --      ->select(e | e.name.toLower().matches('.*users.*'))
                         --works



 -- return LocalizationTraces mappings  inAbstractStates
def: getLocalizationMappingsInAbstractState:Collection(ModelElement) =
      getLocalizationMappings->select(e |
      e.metatypeUri.matches('http://se.cs.toronto.edu/mmint/DLocalisationInStateMachine/DLocalisationInAbstractState'))


 -- return LocalizationTraces mappings  in FiringElements
def: getLocalizationMappingsInFiringElement:Set(ModelElement) =
      getLocalizationMappings->select(e | e.metatypeUri.matches(''))
               ->asSet()




                              ------------Transitive closure Examples Queries
```

```


def: TransitiveClosure : Collection(duide::DDecision)=
    --closure includes the element itself, so I do the closure of the direct dependents
    getDecisions ->select(d | d.question.matches('.*Compromise.*') ).dependsOn.target ->closure(  dependsOn.target )


def: TransitiveClosureOfASpecificType: Boolean =

    getDecisions ->select(d | d.question.matches('.*Compromise.*') )
                                                            .dependsOn
                                                            ->select(oclIsTypeOf(duide::DExcludes))
                                                            ->collect(target)

                                    ->closure( dependsOn ->select(oclIsTypeOf(duide::DExcludes)).target )



--based on statemachine.ecore
def: StatesTransitiveClosureOfASpecificState : Collection(statemachine::State) =
    let s = getStates ->select(name.matches('.*Seeding.*'))in
            s ->closure(transitionsAsSource.target)
        --  ->union(    s-> closure(transitionsAsTarget.source)) I chose to take the closure in just one direction

--based on statemachine.ecore
def: TransitionsTransitiveClosureOfASpecificTransition : Collection(statemachine::Transition) =
    let t = getTransitions ->select(action.matches('.*share.*'))in -- name should be unique
            t ->closure(target.transitionsAsSource)
        --  ->union(    t-> closure(source.transitionsAsTarget))




-- ************************************************    CONSTRAINTS *********************************************

--  *******************  Test  CONSTRAINT


 -- only 1 stateMachine in MID of type StateMachine
def: oneStateMachine() : Boolean =  Model.allInstances() ->collect(EMFInstanceRoot)
                                         ->select(oclIsTypeOf(statemachine::StateMachine))
                                        ->collect(oclAsType(statemachine::StateMachine))
                                        ->size() =1


--  *******************  LOCALIZATION  CONSTRAINTS

 -- 1.All uncertainties should be localized (e.g should at least have one localization trace).         --tested  --done
def: allUncertaintiesAreLocalized() : Boolean =
        --are all element of getUncertainties included in getLocalizationMappings_DUncertaintyEndpoints

     getUncertainties->forAll(

        getLocalizationMappings_DUncertaintyEndpoints->exists( name = 'DUncertainty '+ description )
    )



--2. An uncertainty can’t have two localization traces with the same target element,
--Two Localization traces  can't have same source & target
-- (mappings in MMINT language; not the modelRel level );
def: NoTwoLocalizationMappingsWithTheSameSourceAndTarget: Boolean
=                                    --tested  --done

            not getLocalizationTraces->collect(mappings)
                                        ->exists(
                                    -- = Returns true if self contains the same objects as
                                    *bag* in the same quantities.
                                    --target -asBag should have 2 element (DUncertainty +
                                    modelElem)
                                    l1,l2 | (l1.modelElemEndpoints.target->asBag() =
                                    l2.modelElemEndpoints.target->asBag())
                                                and l1 <> l2
        )



 --  *******************  OPERATIONALIZATION  CONSTRAINTS

 --- 1. A decision that is not resolved should be related to an
 uncertainty.                            --tested  --done
 def: AllNotResolvedDecisionsAreRelatedToAnUncertainty: Boolean =
        getDecisions->forAll(
                (resolved = false)  implies (dUncertainty->size()>0 )
        )


 --- 2.A DRephrasing Dependency only links (sources +  targets) DDecisions with the
 same DUncertainty  --tested  --done
 def: SameUncertaintyForDRephrasing: Boolean =

    getDRephrasingDependencies->forAll(

        --target.dUncertainty.size() = 3
        --asOrderedSet() removes duplicates + orders
        (target.dUncertainty->asOrderedSet()->size() = 1 ) and
        (source.dUncertainty->asOrderedSet()->size() = 1)
        and( target.dUncertainty->asOrderedSet()->first() =
        source.dUncertainty->asOrderedSet()->first() )
    )



 -- 3. Only DPolar (boolean)  decisions, with MAY partiality, can be
 operationalized.                  --tested  --done
def: OnlyDPloarMayDecisionCanBeOperationalized: Boolean =

            --all operationalization mapping endpoints of type decision,
            --(=> operationalized decisions) should have May + DPOLar

      getDecisions->select(d |
         getOperationalizationMappings_DDecisionEndpoints->exists(  name= 'DDecision '+
         d.question ) --LINK EMF TO Elem
                              )
       ->forAll(
      dType.oclIsTypeOf(duide::DPolar)  and allowedPartiality->includes(DPartiality::MAY)
        )


-- 4.A DPolar decision can’t have two operationalization traces with the same target
element,
--no operationalization trace with same source and target.
--in MMINT mapping level not modelRel level
def: NoTwoOperationalizationMappingsWithTheSameSourceAndTarget: Boolean
=                              --tested  --done

            not getOperationalizationTraces->collect(mappings)
                                 ->exists(
                                -- = Returns true if self contains the same objects as
                                *bag* in the same quantities.
                                --target -asBag should have 2 element (DUncertainty +
                                modelElem)
                                op1,op2 | (op1.modelElemEndpoints.target->asBag() =
                                op2.modelElemEndpoints.target->asBag())
                                            and op1 <> op2
        )


-- 5.Every operationalization trace target should be linked to at least one of the
uncertainty
--  localization traces' target elements.
-- e.g, each operationalization trace target should be in the transitive closure
elements
-- of the uncertainty localization traces' target


                         --------------Test Example1
def : getLocalizationTraceTargetsInStatesOfACertainUncertainty: Collection
(ModelElement)  =

    let targetNames =
     getLocalizationTraces->collect(mappings)

                                --selects specific mappings related to the same
                                uncertainty
                              -> select(
                                         modelElemEndpoints
                                        ->collect(target.oclAsType(ModelElement))
                                        ->select(e|
                                        e.metatypeUri.matches('http://geodes.iro.umontrea
                                        l.ca/duide#//DUncertainty'))
                                        ->forAll(e |
                                            e.name.matches(   ('.*complete.*'))
                                            --specific uncertainty
                                                )
                                        )

                                --select the endpoints not related to Uncertainty
                                ->collect(modelElemEndpoints)
                                ->collect(target.oclAsType(ModelElement))
                                ->select(not
                                metatypeUri.matches('http://geodes.iro.umontreal.ca/duide
                                #//DUncertainty') )
                                ->collect( name)    in

    getStates->select( s | targetNames->exists (e | e = 'State '+ s.name))



                        --------------Test Example2
def : getOperationalizationTraceTargetsInFiringElementsOfACertainDecision: Collection
(ModelElement)  =

     let targetNames =
                    getOperationalizationTraces->collect(mappings)

                                --selects specific mappings related to the same
                                uncertainty
                              -> select(
                                         modelElemEndpoints
                                        ->collect(target.oclAsType(ModelElement))
                                        ->select(e|
                                        e.metatypeUri.matches('http://geodes.iro.umontrea
                                        l.ca/duide#//DDecision'))
                                        ->forAll(e |
                                            e.name.matches(   ('.*Bene.*'))--specific
                                            decision
                                                )
                                        )

                                --select the endpoints not related to DDecision
                                ->collect(modelElemEndpoints)
                                ->collect(target.oclAsType(ModelElement))
                                ->select(not
                                metatypeUri.matches('http://geodes.iro.umontreal.ca/duide
                                #//DDecision') )
                                ->collect( name)    in

  getTransitions->select( s | targetNames->exists (e | e = 'Transition '+ s.action))
  --    getStates->select( s | targetNames->exists (e | e = 'State '+ s.name)) --works


                                    --------------Constraint

def: OperationalizationRealtedToLocalization: Boolean
=                                                              --tested --done

            -- 1. only consider  operationalized decisions
        getDecisions->select(d |
           getOperationalizationMappings_DDecisionEndpoints->exists(  name= 'DDecision
           '+ d.question ) --LINK EMF TO Elem
                           )
               --2. for each decision,
            ->forAll(

                      --3.  the operationlization links targets

                      let operationalizationTargetNames =
                      getOperationalizationTraces->collect(mappings)

                                --selects specific mappings related to the same ddecision
                              -> select(
                                         modelElemEndpoints
                                        ->collect(target.oclAsType(ModelElement))
                                        ->select(e|
                                        e.metatypeUri.matches('http://geodes.iro.umontrea
                                        l.ca/duide#//DDecision'))
                                        ->forAll(
                                            name = 'DDecision '+ question   --specific
                                            decision
                                                )
                                        )

                                --select the endpoints not related to DDecision
                                ->collect(modelElemEndpoints)
                                ->collect(target.oclAsType(ModelElement))
                                ->select(not
                                metatypeUri.matches('http://geodes.iro.umontreal.ca/duide
                                #//DDecision') )
                                ->collect( name)    in

                            --4. Operatioanlization transition targets &  state
                            targets
                    let opTransitionTargets =   getTransitions->select( s |
                    operationalizationTargetNames->exists (e | e = 'Transition '+
                    s.action))
                    in let opStateTargets = getStates->select( s |
                    operationalizationTargetNames->exists (e | e = 'State '+ s.name))

                             in

                            --5. the localization link target
                            let localizationTargetNames =
                            getLocalizationTraces->collect(mappings)

                                --selects specific mappings related to the same
                                uncertainty
                                 -> select(
                                         modelElemEndpoints
                                        ->collect(target.oclAsType(ModelElement))
                                        ->select(e|
                                        e.metatypeUri.matches('http://geodes.iro.umontrea
                                        l.ca/duide#//DUncertainty'))
                                        ->forAll(e |
                                            e.name = 'DUncertainty ' +
                                            dUncertainty.description -- the decision's
                                            dUncertainty
                                                )
                                        )

                                --select the endpoints not related to Uncertainty
                                ->collect(modelElemEndpoints)
                                ->collect(target.oclAsType(ModelElement))
                                ->select(not
                                metatypeUri.matches('http://geodes.iro.umontreal.ca/duide
                                #//DUncertainty') )
                                ->collect( name)    in

                                --6. Localization targets in states and the
                                transitions
                let locStateTargets =   getStates->select( s |
                localizationTargetNames->exists (e | e = 'State '+ s.name))
                in let locTransitionTargets = getTransitions->select( t |
                localizationTargetNames->exists (e | e = 'Transition '+ t.action))

                                 in
                                    --7. All op targets should exists in the closure of
                                    the loc targets

                                    --7.1 all opstate targets should exist in the
                                    closure of locStateTargets
                                    -- or in the closure of the states (source &
                                    target) of the locTransitionTargets

                                    locStateTargets->closure(transitionsAsSource.target)
                                    -- I chose to take the closure in one direction
                                    --
                                    ->union(locStateTargets->closure(transitionsAsTarget.
                                    source))


                                        ->union(locTransitionTargets.source->closure(tran
                                        sitionsAsSource.target))
                                    --
                                    ->union(locTransitionTargets.source->closure(transiti
                                    onsAsTarget.source))


                                        ->union(locTransitionTargets.target->closure(tran
                                        sitionsAsSource.target))
                                        --
                                        ->union(locTransitionTargets.target->closure(tran
                                        sitionsAsTarget.source))

                                        ->includesAll(opStateTargets)


                                    --7.2 all optransition targets should exists
                                    inclosure of  locTransitionTargets
                                    -- or in the closure of the transitions related to
                                    locStateTargets
                                    and

                                    locTransitionTargets
                                    ->closure(target.transitionsAsSource)
                                    ->union(
                                    locStateTargets.transitionsAsSource->closure(target.t
                                    ransitionsAsSource) )
                                    ->union(
                                    locStateTargets.transitionsAsTarget->closure(target.t
                                    ransitionsAsSource) )
                                    ->includesAll(opTransitionTargets)
                )




--6  The source DPolar decisions of a DInformationRequirementDependency can't be
operationalized       --tested  --done
--if one of its targets is still unresolved
-- Therefore, the source set can't possibly be operationalized unless the target set
has been already resolved.

def: NotOperationalizedSourceWhenTargetUnresolvedForDInformationRequirementDependency:
Boolean =

       not
       getAllDependencies->select(oclIsTypeOf(duide::DInformationRequirementDependency))
                            -> exists(                  -- there exists a
                            DInformationRequirementDependency ..
                                        target ->exists( not resolved ) -- with one
                                        target that has not been resolved yet ..

                                        and


                                        source ->exists(    -- but one of its DPOLAR
                                        source is operationalized

                                                dType.oclIsTypeOf(duide::DPolar) and


                                                getOperationalizationMappings_DDecisionEn
                                                dpoints->exists(  name= 'DDecision '+
                                                question ) -- operationalized
                                                )

                                            )




--  ******************  DEPENDENCY  CONSTRAINTS

--1. A same set of decisions can not depend on
itself.                                                     --tested  --done
def: NotADependencyWithSameSourcesAndSameTargets  : Boolean =

   --getAllDependencies->asOrderedSet()->first().source->asBag()->oclType()
   getAllDependencies->forAll(
           -- <> Returns true if self does not contain the same objects as *bag* in
```

```
            the same quantities.
            source->asBag() <> target->asBag()
        )



--2.   No Cycle dependencies of the same type
-- e.g .the same set of decisions can not appear in his dependencies transitive closure
, of same type

                        --------------Test Example
-- No cycle dependencies
def: NoCycleDependencies : Boolean
=                                                                   --tested  --done

    not getAllDependencies-> exists(

        source.dependsOn.target ->closure( dependsOn.target   )->includesAll(source)
    )


                        --------------Constraint


def: NoCycleDependenciesOfSameType : Boolean
=                                                         --tested  --done

    NoCycleDependenciesOfDRephrasingType and NoCycleDependenciesOfDRequiresType and
    NoCycleDependenciesOfDExcludesType
    and NoCycleDependenciesOfDInformationRequirementDependencyType



def: NoCycleDependenciesOfDInformationRequirementDependencyType : Boolean =

    not getAllDependencies->select(oclIsTypeOf(duide::DInformationRequirementDependency))
                             -> exists(
                                    target
                                    ->closure(
                                    dependsOn->select(oclIsTypeOf(duide::DInformationRequ
                                    irementDependency)).target )
                                    ->includesAll(source)
            )


def: NoCycleDependenciesOfDExcludesType : Boolean =

        not getAllDependencies->select(oclIsTypeOf(duide::DExcludes))
                             -> exists(
                                    target
                                    ->closure(
                                    dependsOn->select(oclIsTypeOf(duide::DExcludes)).targ
                                    et )
                                    ->includesAll(source)
            )


def: NoCycleDependenciesOfDRequiresType : Boolean =

        not getAllDependencies->select(oclIsTypeOf(duide::DRequires))
                             -> exists(
                                    target
                                    ->closure(
                                    dependsOn->select(oclIsTypeOf(duide::DRequires)).targ
                                    et )
                                    ->includesAll(source)
            )
```

```
def: NoCycleDependenciesOfDRephrasingType : Boolean =

        not getAllDependencies->select(oclIsTypeOf(duide::DRephrasingDependency))
                             -> exists(
                                    target
                                    ->closure(
                                    dependsOn->select(oclIsTypeOf(duide::DRephrasingDepen
                                    dency)).target )
                                    ->includesAll(source)
            )






--3. Can't have more than one dependency between the same two  sets of decisions
def: OneDependencyBetweenSameSets  : Boolean
=                                                          --tested  --done

        not getAllDependencies->exists(
                -- = Returns true if self contains the same objects as *bag* in the
                same quantities.
                dep1,dep2 | (dep1.source->asBag() = dep2.source->asBag())
                            and (dep1.target->asBag() = dep2.target->asBag() )
                            and dep1 <> dep2
        )


-- 4. A set of only DPolar decisions can't be rephrased.
def: ASSetOfDPOLARDecisionsCanNotBeRephrased : Boolean
=                                                 --tested --done

    getDRephrasingDependencies->forAll(

    not source->forAll(

            dType.oclIsTypeOf(duide::DPolar)
        )
    )


-- 5. Logical dependencies can only exist between sets of two DPolar decisions.
def: LogicalDependenciesCanOnlyExistBetweenSetsOfDPolarDecisions : Boolean
=                            --tested --done

        getDLogicalDependencies->forAll(

            source ->forAll(
                dType.oclIsTypeOf(duide::DPolar)
            )

            and

            target ->forAll(
                dType.oclIsTypeOf(duide::DPolar)
            )
        )
```