Why and How to Make Modelling Languages for Requirements Engineering? An Introductory Tutorial [*][‡]

Ivan J. Jureta
Fonds de la Recherche Scientifique – FNRS
and
Department of Business Administration
University of Namur
ivan.jureta@unamur.be

October 26, 2014

Abstract

You have a Requirements Problem (RP) to solve, if you have information about unclear, abstract, incomplete, potentially conflicting expectations of various stakeholders and about the environment in which these expectations should be met, you know that there is presently no system which meets these expectations, and you need to define and document a set of clear, concrete, sufficiently complete, consistent requirements, which are approved by the stakeholders as appropriately conveying their expectations, and will guide the engineering, development, release, maintenance, and improvement of the system which will in fact meet stakeholders' expectations.

An RP is a complex problem solving task, as it involves, for each new system, the discovery and exploration of, and decision making in new and ill-defined problem and solution spaces. Requirements Engineering (RE) is the field concerned with understanding how RPs are solved and how to do so more efficiently and for better results, that is, higher quality systems.

Solving an RP involves various activities, such as elicitation, categorisation, evaluation, prioritisation, negotiation, prediction, and so on, all of which use and produce primarily information. Requirements Modelling Languages (RMLs) are modelling languages used to facilitate such activities, by representing information so that computations can be performed on it, in order to answer questions encountered during problem-solving. RMLs have been a key part of RE research and practice since the earliest days of the field. Various RMLs have been proposed, from those that provide a visual syntax and guidelines for using the visual syntax, but offer no automated reasoning on their models, to more complicated ones, built on top of formal logics.

This tutorial is an introduction to why and how to make RMLs. It covers recurring topics in RML design, all illustrated by defining and moving from simpler to more complicated RMLs. The motivation for the tutorial is to make RML design more accessible, and help understand the challenges involved in designing new and changing existing RMLs.

[*] Go to `http://jureta.net/requirements-modelling-languages` for all tutorial material. Ivan Jureta holds copyright for this text and all tutorial material, except photographs in the slides. Request permission if you wish to use any of the tutorial material in its original or changed form.

[†] This text is part of the material for the tutorial titled "How to make Requirements Modelling Languages?", at the 33rd edition of the International Conference on Conceptual Modelling, Atlanta, GA, held from the 27th to 29th October 2014.

[‡] I am grateful most of all to John Mylopoulos and Alexander Borgida, who were very patient in many discussions that we have had on these topics since 2007. I have co-authored papers on RMLs with them, as well as with Neil Ernst, Alberto Siena, Anna Perini, Angelo Susi, Stéphane Faulkner, Pierre-Yves Schobbens, and many others. They have all influenced the content of this tutorial in one way or another. This does not mean that we agree on the ideas which I present here.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

You have a Requirements Problem (RP) to solve, if you have information about unclear, abstract, incomplete, potentially conflicting expectations of various stakeholders and about the environment in which these expectations should be met, you know that there is presently no system which meets these expectations, and you need to define and document a set of clear, concrete, sufficiently complete, consistent requirements, which are approved by the stakeholders as appropriately conveying their expectations, and will guide the engineering, development, release, maintenance, and improvement of the system which will in fact meet stakeholders' expectations.

If you need to solve RPs repetitively, perhaps in coordination with others and, or are interested in the research on modelling languages that can help solve RPs, then this tutorial should be relevant.

Alternatively, you can see this as a tutorial for professionals and researchers who need to set rules on how to document the inputs, decisions, and outcomes during early phases of system design. You need to set such rules if you are repeatedly involved in situations that have these characteristics:

- there are individuals, called stakeholders hereafter, who have expectations which should be met,
- there is a need to design, and then make a system which should meet their expectations,
- it is necessary to document information about these expectations and design decisions for how to meet them, so that the system can be made to meet these expectations and to evaluate if the system does, in fact, meet them.

Such rules may recommend, for example, how to document the information about the expectations, that is, about requirements, environment constraints, design options for satisfying the requirements, and how to, among others, refine requirements, identify how requirements interact with environment constraints, determine the consequences of these interactions, which design options satisfy which requirements and how well, and so on. In other words, the rules will recommend how to make models, which represent information used in problem-solving during early system design.

The rules are typically used by interdisciplinary teams. These include representatives of disparate groups with various interests. For example, investors may want arguments and predictions about how alternative system design decisions will influence their return on investment. Product designers need information on customer expectations, environment constraints, feasibility evaluations of design options, and so on, in order to make informed design decisions. Engineers from relevant fields (construction, software, hardware, and so on, depending on the purpose of the system) need to evaluate the feasibility of alternative designs and requirements, propose concrete processes and technologies that can satisfy requirements, identify technology or other constraints which may require changes to designs, etc. Government representatives and legal professionals may want to determine if the chosen system design will make the system comply with relevant laws. Business analysts and requirements engineers will need to ensure that the relevant information from the said participants is documented in a clear way, that design options are known, and that it can be demonstrated, from that documentation, that if the system is made and run according to the specification, then it will satisfy its requirements.

The rules should help coordinate these disparate groups and interests towards producing, and agreeing on a system design. They are applied in order to facilitate collaborative problem-solving during design, by clearly representing the information relevant for design, the design options, relative merits of the options, and in order to produce documentation of the adopted design.

The documentation is used, for example, as input to the engineering and development of the system, when choosing subcontractors, for quality evaluation and assurance, to assess compliance to regulations and/or standards. The rules are usually applied in early phases of system design, because they normally do not produce a system design specification which is sufficiently complete and detailed to serve as a blueprint to make the system. Instead, they result in a clear definition of the system's purpose, of the main constraints it has to live with, and of (some) constraints on how it should achieve its purpose. All these are inputs for engineers responsible for producing the specification of the system's detailed design, for which formal methods [27] are more relevant than the kinds of modelling languages discussed in this tutorial.

1.2 Objectives

Using more specialised terminology, this tutorial is on how to make Requirements Modelling Languages (RMLs). RMLs are rules on how to represent and analyse information which is used when solving RPs. RPs are a class of problems studied in

Requirements Engineering (RE). The default view in RE is that there is an RP instance to solve when:

1. there are requirements that a system-to-be should satisfy for its stakeholders,
2. there is information about the environment in which that system will run, and
3. the system needs to be designed, so that it can then be made according to the design.

Solving an RP amounts to producing an specification of a design of the system-to-be. The specification is a representation of the system design. The specification should be such that its makers can demonstrate, that if the resulting system is implemented according to that specification, then that system will satisfy its requirements as well as feasible, within the constraints of the environment.

The tutorial has the following objectives:

- to show how to make new RMLs, through a progression from simple to more complicated ones,
- to review and illustrate major topics and challenges when making RMLs, and
- to discuss, in light of the above, the designs of oft-cited RMLs in RE research.

1.3 Outline

The tutorial has three parts:

- Part 1 runs through Sections 2 to 4, and sets the stage for the tutorial itself:
  - Section 2 gives the general background on Requirements Engineering and Requirements Problems, the role of Requirements Models and Requirements Modelling Languages in problem-solving, and presents the main motives for this tutorial.
  - Section 3 presents the example used throughout the paper. The example is inspired by the problem of designing the London Ambulance Service's Computer-Aided Dispatch (LASCAD) system [2], a case study which is commonly used in RE research. All Requirements Models in this paper represent parts of the information from the example.
  - Section 4 introduces the notion of Language Service, explains how it is used to organise the tutorial, and sets up some naming conventions used in subsequent parts.
- Part 2 runs through Sections 5 to 13. Each section focuses on a recurrent topic and challenge in RML design, and defines new RMLs for illustration:
  - Section 5 focuses on defining relations in RMLs, the relations being over bits and pieces of information used in problem-solving. The discussion revolves around how to define individual relations, issues in defining languages that have many relations, and on three RE concerns, which have usually been addressed via specialised relations in RMLs.
  - Section 6 is on how to embed guidelines for problem-solving into RMLs. Guidelines recommend how to do something in problem-solving.
  - Section 7 looks at how bits and pieces of information in problem-solving can be categorised (for example, as "requirement", "domain knowledge", "specification", "goal", and so on), why this can be useful, and how categories can be defined in RMLs.
  - Section 8 is on how to represent alternative design options in models, critical capability that a language should have, if it is to assist decision-making.
  - Section 9 looks at how to associate variables to model parts, and functions to relations between them, so that the value assigned to one model part depends on the values assigned to other parts. In other words, the focus is on valuation, which makes it possible to ask such questions of models as, for example, if the conditions described in a model part will be satisfied, if conditions described by some other model parts are satisfied as well, the allowed values being "satisfied" and "not satisfied". The section looks at different value types and the combined use of several value types in the same language.
  - Section 10 looks at how to represent that some value assignments in models are uncertain, that is, how to have random variables in models. It shows how models can include independent random variables, and how to work with dependent random variables.
  - Section 11 is concerned with how to represent the relative desirability of value assignments in models, so as to say, for example, that satisfying some requirement is strictly more desirable than satisfying another. The section discusses various kinds of preferences, criteria, and how to use preferences to find the most desirable value assignments in models.
  - Section 12 discusses how models in languages of this tutorial can be mapped to theories in formal logic. The aim is to show one way how RMLs can be related to formal logics, and illustrate why and when that may be interesting, as well as the risks it carries.
  - Section 13 illustrates how to define RP classes, and why and how to relate them to RMLs.

- Part 3 is condensed in Section 14. It uses the terminology and languages introduced in the tutorial, to discuss the designs of well-known RMLs.

2 General Background

This section recalls the usual background to the terms RE (Section 2.1), RP (Section 2.2), RML (Section 2.3), and Specification (Section 2.4), and reiterates common arguments from RE about why RPs and RMLs are important in the engineering of systems in general. The section closes by arguing why it is relevant to make new RMLs (Section 2.5).

2.1 Requirements Engineering

RE focuses on how to elicit, model, and analyse the requirements and environment of a system-to-be in order to design its specification.

It is on the basis of its specification that the system is built, updated, changed, its new releases planned, made, announced, rolled out. The system's scope may be limited to specific (parts of) software and/or hardware, or widened to include such issues as work guidelines, business processes, responsibilities, incentives, contracts, or other concerns.

Specifications can take different forms, from minimalistic to-do lists that hint at stakeholders' expectations and subsume implicit engineering solutions, to elaborately structured documentation on contracts with employees and suppliers, responsibilities of positions in the value chain, guidelines for employee coordination and collaboration, as well as software specifications made using formal methods.

The design of the specification, usually called the RP, is a complex problem solving task, as it involves, for each new system-to-be, the discovery and exploration of, and decision making in new and ill-defined problem and solution spaces.

Difficulties involved in solving an RP instance are illustrated by the variety of topics studied in RE research, such as requirements elicitation [53, 66, 35], categorization [33, 140, 79], vagueness and ambiguity [98, 90, 77], prioritization [82, 9, 65], negotiation [89, 11, 75], responsibility allocation [33, 21, 48], cost estimation [12, 15, 118], conflicts and inconsistency [100, 64, 131], comparison [98, 90, 91], satisfaction evaluation [14, 98, 84], operationalization [51, 48, 44], traceability [54, 107, 28], and change [24, 136, 18].

RE issues are present when designing new and changing existing systems; they are there whatever the system class and domain, and regardless of the extent to which people are involved in the system: from autonomic Internet-scale clouds, to traditional desktop applications, industrial expert systems, and embedded software, all enabling anything from massive mobile apps ecosystems, global supply chains, medical processes, business processes, mobile gaming, and so on. Moreover, RE issues are present regardless of how the software in the system is designed and made, from a military waterfall to a startup's own agile dialect, and from organisations where developers talk directly to customers, to those where product designers, salespeople, or others mediate between requirements and code. In all these cases, there will be information on the requirements and the environment, and it will be necessary to design how the requirements will be satisfied in the given environment.

2.2 Requirements Problems

The *de facto* default view in RE is that the specification is produced *incrementally*, starting from incomplete, inconsistent, and imprecise information about the requirements and the environment, and that each design step reduces incompleteness, removes inconsistencies, and improves precision, towards the specification of the system [13, 33, 56, 100, 46, 140, 130, 21, 111, 76, 44]. This general view of the design process, that we start with less detailed and somehow deficient information, and increase detail and remove deficiencies, is also shared in other domains interested in design, such as architecture [123, 86] and civil engineering [3].

This important and general conceptualisation of the aim in RE is most clearly formulated in Zave & Jackson's seminal paper, "Four dark corners of requirements engineering" [140] and is echoed in discussions on the philosophy of engineering [121]. Their view, denoted ZJ hereafter, is aligned with some of the most influential research in the field, which both preceded and followed the said paper, including, for example, contributions from Boehm et al. [13, 11], van Lamsweerde et al. [33, 34, 131, 132, 130, 90], Mylopoulos et al. [98, 56, 21], Robinson et al. [111], Nuseibeh et al. [100, 71], to name some.

According to the ZJ view, in any concrete engineering project, RE is successfully completed when the following conditions are satisfied [140]:

1. *"There is a set R of requirements. Each member of R has been validated (checked informally) as acceptable to the customer, and R as a whole has been validated as expressing all the customer's desires with respect to the software development project.*
2. *There is a set K of statements of domain knowledge. Each member of K has been validated (checked informally) as true of the environment.*
3. *There is a set S of specifications. The members of S do not constrain the environment; they are not stated in terms of any unshared actions or state components; and they do not refer to the future.*

4. *A proof shows that $K, S \vdash R$. This proof ensures that an implementation of S will satisfy the requirements.*
5. *There is a proof that S and K are consistent. This ensures that the specification is internally consistent and consistent with the environment. Note that the two proofs together imply that S, K, and R are consistent with each other."*

These conditions lead to the following compact formulation of the default problem that one solves in RE; it is called the Default Requirements Problem hereafter.

Definition 2.1. *Default Requirements Problem (DRP): Given* a set R of requirements, and a set K of domain knowledge, *find* a specification S, such that S satisfies the following conditions:

1. There is a proof of R from K and S, written $K, S \vdash R$,
2. K and S are consistent, written $K, S \nvdash \bot$.

2.3 Requirements Models and Requirements Modelling Languages

Representations of information about RP instances, called Requirements Models, are intended to facilitate problem-solving: when requirements are elicited, they are documented in such models; when they are negotiated, the parties involved use models in communication; the models are a basis for estimating costs, risks, and deadlines of alternative designs; they are used to evaluate completeness and clarity of requirements and designs, to determine if we have identified a solution, or more alternative solutions, to rank alternative solutions to the RP, to track the progress of system implementation, and so on.

Requirements Models are made using RMLs. Research on RMLs goes back to the original framework for requirements models, *RMF* [57]. Many different RMLs have been proposed since, including ERAE [42], *NFR* [98], *KAOS* [33], *i** [139], *LQCL* [71], and *Techne* [76].

RMLs have different shapes and forms. *RMF* is a custom formal language with built-in abstraction mechanisms, including aggregation, classification, and generalisation. *KAOS* uses the language of first-order linear temporal logic, and categorises ground formulae as instances of concepts, such as goals, requirements, constraints, while categorising proof patterns as goal refinement, conflict, or other relations of interest when doing RE. *i** has a custom visual notation, which comes together with axioms constraining the making and reading of *i** models. *LQCL* uses the language of classical propositional logic to represent requirements, imposes no classification to requirements, and uses a set of inference rules that are paraconsistent, so that it allows automated reasoning over inconsistent sets of requirements. *Techne* has its own formal language, where expressions are a subset of propositional Horn clauses, with a mechanism to assign types of requirements to facts and clauses.

2.4 Specifications as Solutions

In the DRP, the specifications set S is a representation of the solution to the problem. We will adopt this same stance here, and write Specification to denote the description of the solution to an RP.

The format and content of a Specification depend on the RML used to make it, and of the knowledge of requirements and environment. The RML influences how the information about the solution is represented, while the knowledge of requirements and environment influences the content, that is, the information which is represented.

Specifications made with RMLs are usually not sufficiently complete and detailed, to serve as the blueprint for making the system-to-be. Instead, each is a synthesis of design decisions that are intended to narrow down the purpose of the system-to-be, and impose some constraints on how it should realise that purpose. It is still necessary to subsequently produce detailed specifications that will say how exactly the system should work, in order to realise its purpose within the constraints of its environment. If the system is, or includes software, then formal methods [27, 138] could be used to make detailed specifications. If the system involves making buildings, then architectural drawings, construction and mechanical engineering schemas, and so on, will be parts of detailed specifications. This is apparent from the modelling toolsets of RMLs. An RML will rarely, if ever, include the same conceptual tools, as, for example, formal methods, and therefore, are not intended to replace formal methods, or more generally, specialised languages for the specification of detailed designs.

2.5 Why Make (New) Requirements Modelling Languages?

The relevance of RMLs for describing and solving RPs depends on the influence these languages have on individuals who learn them, when they are thinking about, and solving RPs.

That RMLs do influence thinking during RE, is usually an implicit assumption, as well as an important motivation for the research and teaching on these languages, and on the creation of guidelines, processes, methods for making and manipulating the resulting models.

The assumption is very much related to research on the relationship between language and thought, in linguistics and cognitive science. It is aligned with the Sapir–Whorf hypothesis [83], which is that "[s]tructural differences between language systems will, in general, be paralleled by nonlinguistic cognitive differences" and that "[t]he structure of anyoneâĂŹs native language strongly influences or fully determines the world-view he will acquire as he learns the language". It is related to the linguistic relativism position [59, 49], which is that [102] "use of the linguistic system [...] actually forces the speaker to make computations he or she might otherwise not

make."[1] In cognitive science, there are empirical results [142, 31, 74] supporting the claim that "external representations" (what Requirements Models are in RE) are relevant when solving complex problems, and not only as memory aids, but that they also influence how people discover, describe, and explore problems and their solutions. Similar views were echoed in programming language design, for example, in Kenneth E. Iverson's 1979 Turing award lecture, on notation as a tool of thought [73].

If RMLs do in fact influence how one thinks about and solves RPs, then there are two related motives for learning *how to make* RMLs:

- The practice-oriented motive is to be able to create new RMLs, change and extend existing ones, in order to better solve RPs specific to domains, system classes, projects, organisations, and so on.

- The theory-oriented motive is that trying to teach how to make RMLs makes it necessary to build a body of knowledge on how to relate, extend, compare, and analyse RMLs in a systematic way. Existing research on these topics is sparse [60, 111, 76].

[1] Linguistic relativism is usually related to the nativist position; the latter argues that concepts are prior to and progenitive of natural language. The two positions are usually not seen as conflicting. As Gleitman & Papafragou note [52]: "To our knowledge, none – well, very few – of those who adopt a nativist position on these matters reject as a matter of *a priori* conviction the possibility that there could be salience effects of language on thought. For instance, some particular natural language might formally mark a category whereas another does not; two languages might draw a category boundary at different places; two languages might differ in the computational resources they require to make manifest a particular distinction or category."

3 Example

All Requirements Models in this tutorial represent information from the example in this section. The example draws on the London Ambulance Service's Computer-Aided Dispatch (LASCAD) system [2], which has often been used in RE to illustrate RMLs [71, 131, 132, 90]. The description of the example below borrows Beynon-Davies' presentation of LASCAD [10].

LASCAD was intended to replace manual dispatching of ambulances to incident locations. A manual dispatching system consists of the following [10]:

- *"Call taking. Emergency calls are received by ambulance control. Control assistants write down details of incidents on pre-printed forms. The location of each incident is identified and the reference co-ordinates recorded on the forms. The forms are then placed on a conveyor belt system that transports all the forms to a central collection point.*
- *Resource identification. Other members of ambulance control collect forms, review details on forms, and on the basis of the information provided decide which resource allocator should deal with each incident. The resource allocator examines forms for his/ her sector and compares the details with information recorded for each vehicle and decides which resource should be mobilised. The status information on these forms is updated regularly from information received via the radio operator. The resource is recorded on the original form that is passed on to a dispatcher.*
- *Resource mobilisation. The dispatcher either telephones the nearest ambulance station or passes mobilisation instructions to the radio operator if an ambulance is already mobile."*

The rationale for replacing manual dispatching is that the manual identification of the precise incident location, production of paper-based records, and tracking of ambulance locations were seen as time-consuming and error-prone. Replacing the manual system with a computer-aided one was considered as a way to improve service to patients.

A computer-aided dispatch system would be designed to support the following [10]:

1. *"Call taking: acceptance of calls and verification of incident details including location.*
2. *Resource identification: identifying resources, particularly which ambulance to send to an incident.*
3. *Resource mobilisation: communicating details of an incident to the appropriate ambulance.*
4. *Resource management: primarily the positioning of suitably equipped and staffed vehicles to minimise response times.*
5. *Management information: collation of information used to assess performance and help in resource management and planning."*

The example describes the problem of designing the computer-aided dispatch system, within an environment where dispatching is done manually. In the rest of the paper, **Computer-Aided Dispatch System (CADS)** refers to the system that needs to be designed.

Although there is relatively little of it, the information above is rich: it mentions various activities that dispatching involves (for example, call taking and resource identification), the normal sequence of these activities (call taking precedes resource identification), the organisational positions involved in these activities (control assistants, resource allocators, dispatchers), the responsibilities of the positions (resource allocator decides which ambulance to mobilise), and so on.

4 Preliminaries

The tutorial starts with simple and progresses towards more complicated languages. The difference between languages is described using the concept of Language Service. Simpler languages deliver fewer of these. Section 4.1 explains what Language Services are, and how I use them in modelling language design. Section 4.2 explains the naming convention for all languages in the tutorial.

4.1 Language Services

Let Q be the abbreviation of a question, such as, for example, "Which requirements are satisfied in the given Requirements Model?".

The Language Service Q is a capability of an RML X, which consists of the following: *given a model made with language X, any person who asks the question Q, about that model X, will obtain the same answer.*

I refer to a Language Service by that question Q, which an RML should help its users answer; so I will say "language X delivers Language Service Q". Language Services are central to this tutorial, as they influence the design of all RMLs in it, and the sequence in which I present these RMLs.

The introduction and use of Language Services is motivated by the assumption mentioned in Section 2.5, that an RML should influence how one thinks about and solves RPs. More specifically, I will assume that an RML will effectively do so, if it can *do* something for its user, that is, *if its user can delegate part of the problem-solving effort to the RML.*

Think of it this way: there is a language user, a person who needs to solve an RP, and suppose that there is software, which she uses to make Requirements Models. To find the solution to the RP, as well as to properly formulate the RP to solve, she invests some effort. Problem-solving is the name for what she does.

Part of that effort goes into making and changing the model itself, the *modelling*, and part of it goes into asking questions and finding answers to them, by inspecting the model, the *reasoning*. Such questions can be, for example, "Which requirements in the model cannot be satisfied together?", or "Does the model describe how to satisfy some requirement X in it?", and so on. Now, she can probably find answers to many such questions by having natural-language be her modelling language, and ordinary text her models; she brings the text up on a screen, or prints it out, then searches through it and reads it to find the answer.

But there are two problems with this, if not more. If another person tries to find the answer to the same question, from the same model, what guarantees that the answer will be the same? Yet it should, unless you want models to cause confusion.[2] And if the model gets big – the text is long – will it not become, at some point, too difficult to find answers, and will there not be questions to which you want answers, yet cannot find them within some reasonable time?

To make problem-solving easier, I can add rules on how to make diagrams that represent things, actions, and so on, in the text, and can change the software to enable it to answer questions by doing some processing on the models. The software will then process a model, and return an answer. To abstract from implementation specifics, I will say that *the engineer delegates part of the effort to the RML*, and the RML has to say what its models are, and how to process them to answer questions.

Language Services are used to describe parts of the problem-solving effort, which the engineer can delegate to an RML. If an RML can answer some specific question, then I can define a Language Service, and I will say that that RML delivers that Language Service. Languages can be compared in terms of Language Services that each delivers.

Language Services are not defined as some specific concepts, relations, rules, or algorithms that are part of a language. It follows that two languages may be said to have the same Language Service, even if they have very different components and work in different ways to answer the corresponding question.

4.2 Naming Conventions

Every language defined in this tutorial has two names. One is its so-called module name and the other is its common name.

The module name lists the abbreviations of all modules in that language. Section 5.2 explains what a language module is. For now, it is enough to know that a module is a self-contained part of a language, which can appear in more than one language. That is, it can be reused when making different languages.

For example, a module name for one of the language in Section 5.3 is L.(F, r.inf.pos, r.inf.neg, f.map.abrel.g). This says that the language is made of four modules, denoted by F, r.inf.pos, r.inf.neg, and f.brel2g. Each language module in the paper has a unique abbreviation, and those abbreviations are used to form the module names

[2] Any model probably can be read in different ways by different people, but it is feasible, when making models that have to answer very specific questions, to make sure that they do not give confusing answers *to those questions.* If one writes $x+5=7$, and says to another that these are numbers of apples, the other might debate if they are of the Granny Smith or Golden Spire variety, but both would answer 2 if asked for the value of x.

of languages. The point is to know what modules a language includes, simply by looking at the name.

The common name has nothing to do with the module name of a language, in that neither is inspired by the other. The common name is chosen simply to make it easier to refer to a language, when the module name is unnecessary. Common names are the common names of navigational stars in celestial navigation, taken from the Nautical Almanac [72].

Appendix A lists all language modules and languages defined in this tutorial, with their module names and common names.

5 Relations

Overview and Motivation

This section is on how to define *relations* over bits and pieces of information used in problem-solving. The discussion revolves around how to define individual relations, issues in defining languages that have many relations, and on two RE concerns, called influence and rationale below, which have usually been addressed via specialised relations in RMLs. More specifically, the section is on:

1. How to represent in Requirements Models that we start design with less detailed information, and incrementally add details to it? (in Section 5.1),
2. How to define oft-needed relations in such a way, that they can be reused when defining new RMLs? (Section 5.2),
3. How to represent that satisfying some requirements influences the satisfaction of others? (Section 5.3),
4. How to represent the rationale for design decisions? (Section 5.4), and
5. If an RML includes several relations, then how to avoid errors in using these relations together? (Section 5.5).

Problem-solving in RE involves working with information, obtained through interviews, observation, simulation, role-playing, from documentation, through reflection, creativity, and so on. You need to organise this information in order to understand the concrete problem to solve, to design its one or alternative solutions, compare them, and do all else that might be necessary, in order to produce a solution (Section 2.1 mentioned some of many potential tasks).

You can organise this information by making representations of it, splitting representations into pieces, and stating relations over the pieces. The first part of the tutorial focuses on how you can define relations in RMLs, so that their models can represent instances of these relations over pieces of information. In turn, relations let you reconstruct, from the pieces, your initial understanding of the initial whole, and also, to identify interactions between these pieces, which was not feasible when they were not split up.

There are two practical reasons to start the tutorial by focusing on relations *only*, and so have only one category of information. Firstly, I can postpone the discussion of such issues as, when a piece of information should be called a requirement, a goal, a task, a specification, or otherwise, that is, the issue of categorisation, to which I return in Section 7. [3] Secondly, committing already now to some specific categories would bias the discussion to a specific class of RPs. This is because RP classes come with their own information categories: in DRP, for example, they are "requirement", "domain knowledge", and "specification". There is no need to privilege one RP class over others this early in the tutorial.

5.1 How to Define a Language with One Category and One Relation?

As usual, a relation R over some sets $X_1, \ldots, X_n$ of things, be they requirements, laws, (or representations of) people, cars, buildings, or clouds, of same or of different kinds, is a subset of the Cartesian product of these sets, that is, $R \subseteq X_1 \times \ldots \times X_n$. A relation is used to indicate that the things it relates share the property which the relation stands for. For example, if *in love with* denotes a binary relation over people, and people are identified by first names, then *Pierre in love with Marie* is an instance of the *in love with* relation, and is intended to convey that they share the property that we conventionally understand as Pierre being in love with Marie, and that that Marie is the person whom Pierre is in love with. With this simple idea in mind, consider the following exercise. The rest of this section shows one way to solve it.

Exercise 1: Define a language which has one category and one relation

Define the simplest RML which lets you show that information about requirements increases incrementally as system design progresses. By simplest, I mean something that is easy for others to understand. It can help to consider the following questions.

- What is, or are the Language Services that this language should deliver? Why? Define them.

[3] While I am discussing relations before categories, I do not suggest, for example, that in general, relations should be the primitives in RMLs. I already introduced the notion of Fragment as a primitive, and Fragments are not relations. Also, when I start introducing more categories later, I do not define categories only in terms of relations. So I am not saying, for example, that pieces of information are in relations *because* they satisfy some monadic properties first and foremost, and that them having these properties influences the relations in a language. For example, this amounts to saying that it is is because there are things called requirements and others called specifications, that I am interested in relations that indicate how doing according to specifications influences if we satisfy requirements. The opposite approach, where relations are primitives, would be to say that I have to distinguish categories of information that describe what to satisfy (requirements), from those on what to do (specifications), because I am interested in relations that reflect correlation of satisfaction.

- How would you represent that information increases? Try with a relation.
- What is the domain of that relation, what is the relation over?
- How should relation instances read informally? What is it that they should be saying to other people who are using models in that language?
- What are the formal properties of that relation? Is it, for example, transitive?

5.1.1 Choosing a Language Service

I will start by choosing the Language Service which the language should deliver. To do this, recall, from Section 2, that the default view in RE is that RPs are solved incrementally, moving from incomplete or otherwise deficient information, towards less deficient information that describes the problem and its solution.

At each iteration, I want to add information to the model. This new information may be adding details to the information already there. The additional detail may come from explaining how to satisfy some requirement, that satisfying a requirement involves satisfying several more specific requirements, making a requirement less ambiguous, and so on. The same applies to any information in the model, be it requirements or otherwise (such as domain knowledge and specifications in the Default RP).

It is relevant have models which show how information was added during design. Discovery and indecision in problem-solving are two reasons for this, among others.

- *Discovery* refers to starting with relatively little, and progressively increasing knowledge of, for example, the relevant requirements and domain knowledge, their relative importance, their completeness, about ways to satisfy requirements, and so on. At a given time during problem-solving for the CADS, you may not know the various possible ways to identify the incident location; as you learn more about them, you would be adding more details about them to the model.
- *Indecision* refers to the unwillingness, at some time in problem-solving, to commit to, for example, resolve some conflict between requirements in one way and reject all alternative ways to do so, to give a particular interpretation to an ambiguous requirement, or to some specific way of satisfying a requirement. For example, you may decide not to describe in the model a process for choosing the ambulance to dispatch, until you have interviewed the control assistants who have experience in that task.

As I proceed with discovery and postpone commitments, I am adding more detailed information to the model. Instead of deleting the less detailed information when this happens, it is relevant *to keep both in the model.* More specifically, it is useful to indicate in the model which information adds details to which other. Doing so results in a record of *what* I am adding details to and *why* I am adding the more detailed information in the first place.

Modelling the increase in information in a model raises a number of design challenges and is related to many Language Services that various well-known RMLs deliver. For example, in *KAOS*, the ability to answer "Which requirements are more detailed than (that is, refine) the given requirement?"; in *i**, to answer "Which tasks are more detailed than (decompose) the given task?". This leads me to the following Language Service for the new RML. Let x and y be parts of a model M in that language.

Language Service

AddsDetails: Does x add information to y in M?

The Language Service does not define exactly what the model or its parts are, and thereby remains independent of a particular language.

5.1.2 Models over Arbitrary Representations of Information

Natural language text is an accessible and neutral way to represent information about RPs and their solutions, because, respectively, there is no need for additional learning to use it, and it comes with no rules on how to represent, categorise, or work with that information.

If natural language is a casual means of requirements representation, then does ordinary text as a means of representation deliver s.AddsDetails? Consider the following pieces of information, called *Fragments*, about the CADS.

> Emergency calls are responded to. (AddRepEm)
>
> Receive emergency calls. (RecEmCal)
>
> Switch emergency calls to ambulance dispatch centre. (SwtchCal)
>
> No calls are dropped because of timeout. (NoDropCal)
>
> Identify the incident location. (IdIncLoc)
>
> Check if double location. (ChkDblLoc)

Fill out the incident report. (FillIncRep)

Fill out incident report form via software. (FillSwIncRep)

Above, Fragments are ordinary sentences, with an abbreviation for easier referencing. I impose no rules about, for example, how to decompose and combine Fragments. Later, I will in some languages. Moreover, while Fragments can be representations of *propositions*[4], not all of them are: questions arise during problem-solving, and while they cannot be propositions [125, 50, 129] (What do you answer to "Is that question Q true or false?"?), it is relevant to have a record of them, and inevitably, then, have them in models.

But Fragments need not only be parts of natural language text. Datasets, diagrams, photographs, videos, can all be representations of requirements, or of other information which is relevant when defining requirements, solving conflicts between them, getting stakeholders to approve requirements, and so on [32, 109].

Consequently, *a Fragment is any available representation of information, as long as the model user judges it to be relevant for problem-solving in RE.* This is important to keep in mind, as all languages in the rest of this tutorial create models over Fragments. While the present format makes Fragments in natural language text the easiest to use, there is nothing in the languages defined here, which restricts Fragments to text only.

Example 5.1. The CADS example suggests that addressing each emergency involves (at least) taking the emergency call, identifying the incident location, and so on.

It follows that RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, FillIncRep, FillSwIncRep describe one way of satisfying AddRepEm.

AddRepEm thus looks to be *less detailed than* every one of the former statements. Equivalently, AddRepEm is *more abstract than* each of these statements. Also, each of the latter statements is *more concrete, or more detailed than*, and *adds details to* AddRepEm. FillSwIncRep is one of some alternative ways of doing FillIncRep, which makes FillSwIncRep more detailed than, and adding detail to FillIncRep.

The following paragraph summarises this.

> RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, and FillIncRep describe what to do, in order to satisfy AddRepEm. Each informs AddRepEm. FillSwIncRep adds details to FillIncRep, because it describes one way of satisfying FillIncRep.

If you replace each abbreviation above with the corresponding Fragment, you get an ordinary paragraph of text. •

[4] I take McGrath's view on propositions [95], so that they are "sharable objects of the attitudes and the primary bearers of truth and falsity. This stipulation rules out certain candidates for propositions, including thought- and utterance-tokens, which presumably are not sharable, and concrete events or facts, which presumably cannot be false."

Taken as a representation of information about ambulance dispatching, the paragraph in Example 5.1 is subject to no particular rules which would influence how you and I represent and communicate about differences in detail. For example, the paragraph can be seen as a single Fragment, or multiple Fragments, neither of which can be unambiguously established by looking at it alone.

To be able to find the same answer to s.AddsDetails, you and I need to agree on at least two rules, on (i) how to distinguish between Fragments, and (ii) how to record that one adds details to another. Once we do, the result is that we will be no longer documenting our communication about the adding of details using unconstrained text, but text that has to satisfy the new rules. Since these rules are specific to s.AddsDetails, I will call the resulting representations *models*.

5.1.3 A Trivial Modelling Language

One way to distinguish Fragments is to visually separate them. You can write each in a different paragraph. For referencing, you could have a unique identifier for each paragraph.

To record which Fragments add information to others, you and I can agree to write sentences in this format "x informs y", where we replace x and y with relevant Fragment identifiers.

"x informs y" reflects the conclusion of comparing two Fragments x and y, and concluding that x adds information about y. In other words, saying "x informs y" equates to stating a relation between x and y, and begs the question of what properties this relation has.

It makes no sense to say that "x informs x", so the relation is irreflexive. It is also not the same to say that "x informs y" or that "y informs x"; it is one or the other, so that the relation is antisymmetric. It is also transitive, as the following seems reasonable: if you say that x informs y, and that y informs z, then you are also saying that x informs z. Finally, I will not be saying which Fragment informs another, for every pair of Fragments. I might do it for some Fragments only. In conclusion, the "informs" relation on a given set of Fragments is a strict partial order relation.

This gives a language which delivers s.AddsDetails. The language is called L.D1, and is defined only by the rules which you and I agreed on so far:

> Every model M in L.D1 is a graph $(X, \text{r.ifm})$, where:
>
> 1. every Fragment in X is a node,
> 2. every edge is an instance of r.ifm over X,
> 3. r.ifm is a strict partial order on members of X, and
> 4. $(x, y) \in$ r.ifm reads "Fragment x adds details to Fragment y".
>
> L.D1 delivers the following Language Services:

- s.AddsDetails: Yes, iff there is a path from x to y in the transitive closure of M, no otherwise.

Using L.D1 takes me from natural language to a controlled language, and in the process restricts considerably what I can say about why some Fragments add detail to others, for example. This is apparent by comparing models in Example 5.1 and Example 5.2; the latter was made using L.D1 on Fragments in the former example.

Example 5.2. The graph $G = (X, \text{r.ifm}(X))$ is a model in L.D1, where:

- X = { RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, FillIncRep, AddRepEm, FillSwIncRep} is the set of all Fragments,
- The set of edges is this set of r.ifm instances:

$$\begin{aligned}\text{r.ifm}(X) = \{\quad & (\text{RecEmCal}, \text{AddRepEm}), (\text{SwtchCal}, \text{AddRepEm}),\\ & (\text{NoDropCal}, \text{AddRepEm}), (\text{IdIncLoc}, \text{AddRepEm}),\\ & (\text{ChkDblLoc}, \text{AddRepEm}), (\text{FillIncRep}, \text{AddRepEm}),\\ & (\text{FillSwIncRep}, \text{FillIncRep})\ \}\end{aligned}$$

Figure 1 gives a visualisation of this model. The visualisation shows a graph, where nodes are Fragments, and edges labeled "D" are r.ifm instances. •

The main point of Language Services is that *if I made the model in Example 5.2, and gave it to you, and you know L.D1, then you would not need to ask me for my answer to s.AddsDetails, since you can get to the same answer as I.* That is, L.D1 delivers s.AddsDetails.

As an aside, observe that L.D1 cannot be used to solve the Default RP. Delivering s.AddsDetails is not enough, as other Language Services are needed. If you consider that a language is not an RML if it cannot be used to solve the Default RP, then L.D1 is not one.[5] L.D1 models cannot be used to answer seemingly simple questions, such as which of all the Fragments are the most detailed (that is, no other Fragments add detail to them). L.D1 does have important limitations, but the tutorial needs to start from something simple.

5.2 How to Define Languages with Modules?

I defined a simple language in response to Exercise ex:one-category-one-relation. What if I wanted to define new languages, perhaps many of them, all of which would reuse the relation r.ifm in the same way as L.D1? The challenge is summarised in the following exercise.

[5] *i**, for example, also fails this criterion, but is considered an RML. There is, to the best of my knowledge, no widely-accepted set of criteria for when a modelling language is also an RML, despite some suggestions [140, 60, 79, 76].

Exercise 2: Define a relation as a reusable module

Define the relation r.ifm in such a way that it is independent of the syntax of L.D1, and that it can be reused when defining another language, which may have an entirely different syntax, symbolic, visual, or otherwise, than L.D1. You can consider the following more specific questions.

- What are the necessary parts of a definition of a relation? What does one need to know about a relation, in order to use it in modelling?
- When defining a relation, do you consider it necessary to define how its instances are represented? Must a definition of a relation define also the syntax for representing its instances?
- What should be omitted from a definition of a relation, if it is to be reused in different languages?

To solve the exercise, go back to L.D1 and r.ifm, and consider what had to be decided and put into the definition of that relation. I defined r.Inform by answering the following questions:

- What is the name of the relation?
- How a person should read its instances?
- What is its domain?
- What is its dimension (arity)? Is it unary, binary, ternary, n-ary?
- What are its formal properties? More generally, what properties does it have to satisfy?
- Which Language Services I want to deliver with it?

I will answer the same questions for all relations in this tutorial. Hence the Language Module template for relations. Slots in it reflect the questions. Below, it is filled out for r.Inform.

Relation
Inform (r.ifm)

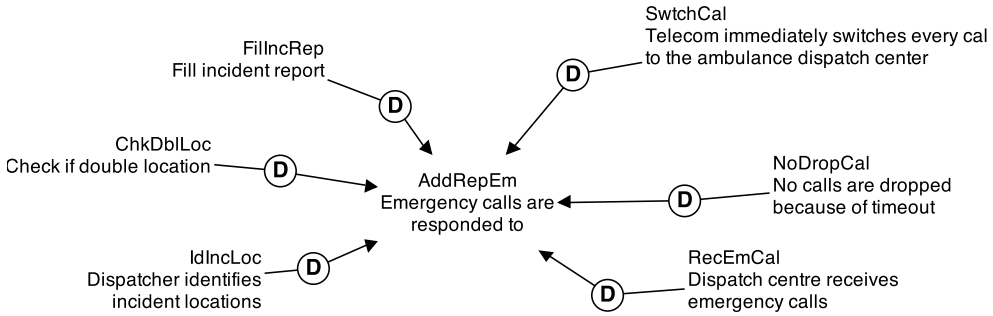


Figure 1: A visualisation of a model in L.D1.

Domain & Dimension

r.ifm $\subseteq$ F $\times$ F, where F is a set of Fragments.

Properties

irreflexive, antisymmetric, and transitive.

Reading

$(x, y) \in$ r.ifm reads "x adds information to y".

Language Services

- s.AddsDetails: Yes, if $(x, y) \in$ r.ifm is in M.

There is a slot for the domain and dimension. Properties are the rules that all relation instances have to satisfy. If some relation r.rel is irreflexive, then you have an error in the model, if it includes $(x, x) \in$ r.del. The properties slot will include all sorts of rules about relation instances, not only common formal properties (as above). Hence the slot's generic name. The "reading" slot says how to read an instance of the relation.

The template includes the abbreviated relation name, r.ifm above. I usually use that abbreviation to refer to the relation, or in general, to Language Module names in the tutorial.

Exercise 3: Define the language which has only r.ifm

Define the simplest language whose models can represent instances of r.ifm. The language definition should not redefine, or repeat the properties of r.ifm.

The template shown with r.ifm focuses on the relation alone. It tries, as much as feasible, to avoid other concerns. For example, it is silent about how sets of relation instances should or could be represented, as sets of symbols denoting relation instances, as graphs where edges denote relation instances, or in some other way. The template avoids issues related to the syntax of the language. When I want to use a relation in a language, I will simply use the name of the relation, and leave its definition in its own module, rather than repeat it in the language definition. Below is the definition of a language which does the same as L.D1.

Language

Alpheratz

Language Modules

F, r.ifm

Domain

Set F of Fragments and r.ifm $\subseteq$ F $\times$ F.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lll} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & (\alpha, \alpha) \\ \phi & ::= & \alpha \mid \beta \end{array}$$

Mapping

$\mathscr{D}(\alpha) \in F$ and $\mathscr{D}(\beta) \in$ r.ifm, that is, every α represents a Fragment from F and every β an instance of r.ifm.

Language Services

Same as r.ifm.

The template has the common name L.Alpheratz and the module name (F, r.ifm). This follows the conventions in Section 4.2. The module name is in parentheses and says that the language takes Fragments and r.Inform. There is the symbolic syntax, defined using BNF notation. You can define it otherwise if you prefer.

I follow Harel & Rumpe [63] on syntax and semantics, and there are consequently slots for syntax, semantic domain, and a function which maps elements of the former to those of the latter. The function is denoted $\mathscr{D}$ in all languages in this tutorial, but its definition is always local to a language. The example below gives a model in L.Alpheratz.

Example 5.3. The following is a model in L.Alpheratz:

$$\begin{array}{lll} M & = \{ & \text{RecEmCal, SwtchCal, NoDropCal, IdIncLoc,} \\ & & \text{ChkDblLoc, FillIncRep, AddRepEm, FillSwIncRep,} \\ & & \text{(RecEmCal, AddRepEm), (SwtchCal, AddRepEm),} \\ & & \text{(NoDropCal, AddRepEm), (IdIncLoc, AddRepEm),} \\ & & \text{(ChkDblLoc, AddRepEm), (FillIncRep, AddRepEm),} \\ & & \text{(FillSwIncRep, FillIncRep) } \} \end{array}$$

M includes individual Fragments and the pairs are instances of r.ifm. •

For every model of L.Alpheratz, you can make a corresponding graph. The graph can be a visualisation of the model, but more importantly, it can be used to compute answers to new Language Services, such as those in the following exercise.

Exercise 4: Find most and least detailed Fragments

How would you change L.Alpheratz, to deliver these Language Service:

- **s.MostDetails**: Which Fragments in M are the most detailed?
- **s.LeastDetails**: Which Fragments in M are the least detailed?

Suppose that $G(M)$ is the graph where every α from M is a node and every $\beta = (x, y)$ is an edge directed from x to y. Let $Cl(G(M))$ be the transitive closure of that graph. You can then deliver s.MostDetails and s.LeastDetails as follows:

- s.MostDetails: All nodes in $Cl(G(M))$ which have no incoming edges.
- s.LeastDetails: All nodes in $Cl(G(M))$ which have no outgoing edges.

There are well-known algorithms for finding transitive closures of directed acyclic graphs, and for finding paths in them [1, 6]. Use them to compute answers to the Language Services above.

In the rest of the tutorial, I define the translations from one syntax to another, or other transformations of models, via Language Modules. These Language Modules are functions, taking (parts of) models as input, making changes, and producing new models or otherwise. When I suggested above that you can make a graph from r.ifm and do computations on those graphs, the more general point is that you may want a language to deliver the following Language Service.

Language Service

RelGraph: What graph is induced by the relation r.R over Fragments in F?

Below is the definition of a function which takes a binary relation and returns a labelled directed graph. It delivers s.RelGraph.

Function

Map a binary relation to a graph
(f.map.abrel.g)

Input

Set F of Fragments and a binary antisymmetric relation r.R $\subseteq$ F $\times$ F.

Do

Let G(F, r.R) = (N, E, l_N, l_E) be an empty labelled directed graph. For every Fragment $f_i \in$ F, add a node n_i to N and let the Fragment label the node, $l_N(n_i) = f_i$. For every relation instance $(f_i, f_j) \in$ r.R, add an edge $(n_i, n_j) \in E$ to the graph, and label the edge r.R.

Output

G(F, r.R).

Language Services

- s.RelGraph: G(F, r.R).

A language which would deliver s.MostDetails and s.LeastDetails would also need additional functions which traverse the graph, and return the sink and source nodes.

The more general point is that templates such as the above promote a modular definition of languages. The template for functions is self-explanatory, giving the inputs, the actions to take on these inputs, the result of those actions, and the Language Services of interest.

You can also have templates for families of languages. You can define analogous languages to L.Alpheratz for many other antisymmetric binary relations in this tutorial. The template for all these languages is as follows, where R is the name of the relation. I added the function f.map.abrel.g, which enables these languages to deliver more Language Services than L.Alpheratz could. I will not spend much time with such languages, as you can define them with the template below.

Language

Alpheratz(R)

Language Modules

F, r.R, f.map.abrel.g

Domain

Set F of Fragments and r.R $\subseteq$ F $\times$ F.

Syntax

Same as in L.Alpheratz.

Mapping

$\mathscr{D}(\alpha) \in$ F and $\mathscr{D}(\beta) \in$ r.R.

Language Services

s.AddsDetails, s.RelGraph, s.MostDetails, s.LeastDetails.

I illustrated above how to define a relation as a Language Module, and then use this module in a language. Sections 5.3 and 5.4 define several other relations. They are all inspired by well-known ideas such as, say, refinement in programming and

correlation in statistics, which are not specific to RE, as well as relations that are central in well-known RMLs. The aim is to give more examples of the modular definition of relations, and then combine these sample relations into new languages in Section 5.5.

5.3 How to Define Different Kinds of Influence Relations?

A recurrent concern in RE is to represent that *satisfying some x has consequences on satisfying some other y*. (x and y may be one or more requirements, domain knowledge, specifications, or otherwise; their categorisation does not matter at the moment.) Satisfying abbreviates "successfully doing what x describes", or if you prefer making it clear that these are models of hypothetical actions, conditions, and such (precisely because they are representations), then it abbreviates "as-if what x describes is successfully done".

This capability is critical for solving the Default RP, for example, since both conditions in that problem are about how the satisfaction of domain knowledge and specifications influences the satisfaction of requirements.

Satisfying some x in a Requirements Model can be independent from the ability to satisfy some other y in the same model. If it is not, then the idea is to have an influence relation between x and y. This relation can indicate positive or negative influence, and various relations have been proposed to do so [111].

Exercise 5: Define one or more relations to represent influence

Define a relation which represents that the satisfaction of a Fragment depends on the satisfaction of another Fragment. What kinds of influence can there be between Fragments? If there are different kinds of influences, would you define a new relation for each? What if you wanted to represent that a Fragment's satisfaction depends more strongly on the satisfaction of some Fragment x than that of some Fragment y?

You can think of satisfaction as being a value assigned to a Fragment. Let *SatVal* denote the satisfaction value of a Fragment, and suppose that $\mathcal{V}$ is the set of all allowed satisfaction values, so that $SatVal: X \longrightarrow \mathcal{V}$, where X is a set of Fragments. There should be an influence relation from x to y iff $SatVal(x) = f(\ldots, SatVal(y))$, that is, if the satisfaction value assigned to x is function of, among others, the value assigned to y.

Due to discovery and indecision in problem-solving, I may incrementally be finding out, or making decisions about the exact function $SatVal(x) = f(\ldots, SatVal(y))$. To be able to represent partial information about influence, I will define several types of influence relations. Some of them will require that I know very little about how $SatVal(x)$ is sensitive to changes of $SatVal(y)$, while others may require that I know more, such as the direction and perhaps strength of that influence.

5.3.1 Presence of Influence

The first influence relation can be used when you know only that a function $SatVal(x) = f(\ldots, SatVal(y))$ does or should exist. The corresponding Language Service is as follows.

Language Service
DoesInfluence: Does the satisfaction of x influence the satisfaction of y in M?

Exercise 6: Define the relation which delivers s.DoesInfluence

Define a relation which conveys only that the satisfaction of a Fragment somehow influences the satisfaction of another Fragment. Whether this influence is positive or negative, or is stronger or weaker than the influence of another Fragment, is not relevant in this exercise.

To deliver s.DoesInfluence, you need a relation for influence. It can be defined as follows.

Relation
Influence (r.inf)
Domain & Dimension
r.inf $\subseteq$ F $\times$ F, where F is a set of Fragments.
Properties

irreflexive and transitive.

Reading

$(x, y) \in$ r.inf reads "the satisfaction of x influences the satisfaction of y", or equivalently, "there is a function $SatVal(y) = f(\ldots, SatVal(x))$".

Language Services

- s.DoesInfluence: Yes, if $(x, y) \in$ r.inf is in M.

Example 5.4. In Example 5.1, the Fragments RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, and FillIncRep described parts of what needs to be done in order to satisfy AddRepEm. This suggests the following r.Influence instances:

(RecEmCal, AddRepEm), (SwtchCal, AddRepEm), (NoDropCal, AddRepEm),
(IdIncLoc, AddRepEm), (ChkDblLoc, AddRepEm), (FillIncRep, AddRepEm).

Let L.Alpheratz_Influence be a language made using the template L.Alpheratz_R from Section 5.2, and r.Influence. Let M be a model in that language, which includes all influence relation instances above and all the Fragments that these instances relate. The corresponding graph is shown in Figure 2. For brevity, edges are labeled "I", rather than "r.inf". •

The example illustrates that it is only necessary to assume that *there exists* a function f such that $SatVal(y) = f(\ldots, SatVal(x))$. When this is done, it is not necessary to also know how exactly the satisfaction of y depends on that of x. It is also not necessary to define the set $\mathcal{V}$ of allowed satisfaction values. This is useful when that set is still unknown or undecided in problem-solving.

5.3.2 Direction of Influence

While you may not know exactly how the satisfaction of y depends on that of x, you may know, or wish to hint that the correlation of their satisfaction values is positive or negative. That is, you want to deliver the following Language Services:

- **s.PosInfluence**: Does satisfying x influence positively the satisfaction of y in M?
- **s.NegInfluence**: Does satisfying x influence negatively the satisfaction of y in M?

Exercise 7: Define one or more relations that deliver s.PosInfluence and s.NegInfluence

Do you need one relation, or more to deliver s.PosInfluence and s.NegInfluence? If more, then how are they different? Or could you define one influence relation, whose parameter would define the direction of influence?

To deliver s.PosInfluence and s.NegInfluence, I define a new relation which can indicate positive or negative influence. I define it as an influence relation that has a parameter. The parameter gives the direction of influence.

Relation

Influence.d (r.inf.d)

Domain & Dimension

r.inf.d $\subseteq$ F $\times$ F, where F is a set of Fragments.

Properties

irreflexive and transitive.

Reading

d is either "pos" for positive or "neg" for negative, and therefore

- $(x, y) \in$ r.inf.pos reads "the satisfaction of x positively influences that of y",
- $(x, y) \in$ r.inf.neg reads "the satisfaction of x negatevely influences that of y".

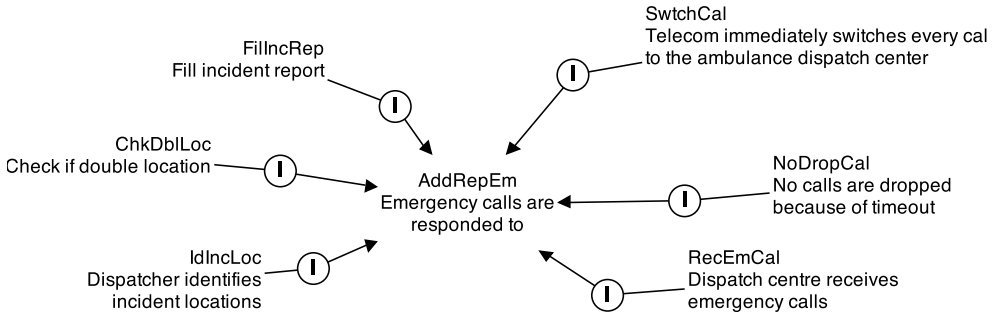


Figure 2: Visualisation of a model in L.Alpheratz_Influence.

Language Services

- s.PosInfluence: Yes, if $(x, y) \in$ r.inf.pos is in M.
- s.NegInfluence: Yes, if $(x, y) \in$ r.inf.neg is in M.

Example 5.5. How would you define a language that can represent both positive and negative influence relations over Fragments? How would you define it by making minimal changes to the definition of L.Alpheratz? The language L.Ankaa below does this.

Language

Ankaa

Language Modules

F, r.inf.pos, r.inf.neg, f.map.abrel.g

Domain

Set F of Fragments. r.inf.pos and r.inf.neg are both over Fragments, so that r.inf.pos $\subseteq$ F $\times$ F and r.inf.neg $\subseteq$ F $\times$ F.

Syntax

Same as L.Alpheratz.

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in$ F, β symbols denote r.inf.pos or r.inf.neg instances, $\mathscr{D}(\beta) \in$ r.inf.pos $\cup$ r.inf.neg.

Language Services

s.PosInfluence, s.NegInfluence.

Figure 3 shows a graph made by merging G(F, r.inf.pos) and G(F, r.inf.neg) made from the same model M in L.Ankaa. The graph shows positive and negative influence relation instances. Positive influences are labeled with "+" and negative with "-". Note that the merge of G(F, r.inf.pos) and G(F, r.inf.neg) could be a hypergraph, since L.Ankaa lets me have positive and negative influence relation instances between same Fragments. •

The difference between $(x, y) \in$ r.inf and $(x, y) \in$ r.inf.d (whichever d is) reflects a difference in the information available about the satisfaction of x and of y. While

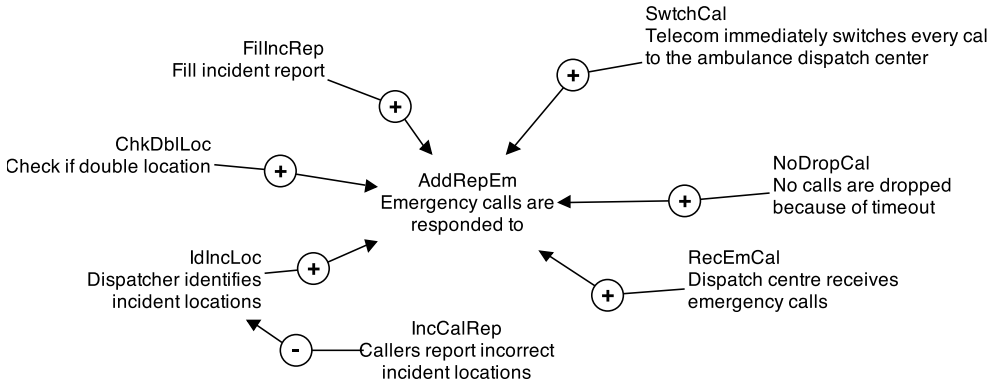


Figure 3: A visualisation of a model in L.Ankaa.

$(x, y) \in$ r.inf says simply that I believe that satisfying x somehow influences satisfying y, $(x, y) \in$ r.inf.d says that I have decided the direction of influence.

5.3.3 Relative Strength of Influence

If you have information about how strongly the satisfaction of a Fragment influences that of another Fragment, this information cannot be represented in models which can show that there is influence, and, or, the direction of influence. The following exercise summarises the problem.

> **Exercise 8**: Represent strength of influence between Fragments
>
> Define a relation, or otherwise, which can be used to convey that the satisfaction of a Fragment more or less strangly influences that of another Fragment. Can this be done with a new relation? What is the scale for strength of influence? Is it absolute or relative? If relative, then what is it relative to?

I will consider the case when the strength of influence of a Fragment on some Fragment x, is relative to the strength of influence of all other Fragments which also influence x.

Suppose that the satisfaction of y is influenced by the satisfaction of several other Fragments $x_1, \ldots, x_n$. How would you indicate that some of them have stronger influence on the satisfaction of y than others? That is, how would you deliver the following Language Service?

Language Service

InfStrength: If the satisfaction of each of $x_1, \ldots, x_n$ influences the satisfaction of y in M, then is the satisfaction of y more sensitive to the satisfaction of x_i than to the satisfaction of x_j, where $x_i, x_j \in \{x_1, \ldots, x_n\}$?

s.InfStrength is about the relative strength of influence. To deliver it, it is necessary to compare the strength of influence of satisfying each $x_1, \ldots, x_n$ on the satisfaction of y. If you knew the exact function $SatVal(y) = f(SatVal(x_1), \ldots, SatVal(x_n))$, then this would not be difficult to do. You could compare the covariance of each x_i to y.

I need a new relation to say that x_i has stronger influence on the satisfaction of y than some x_j. The new relation cannot be over Fragments, because it does not compare Fragments, but the strength of their influence on y. So the new relation, call it r.Stronger_Influence, is over instances of r.inf or those of r.inf.d.

Relation

Stronger influence (r.str.inf)

Domain & Dimension

r.str.inf $\subseteq R \times R$, where R is one of r.inf, r.inf.pos, r.inf.neg.

Properties

irreflexive, antisymmetric, and transitive.

Reading

$((x_i, y), (x_j, y)) \in$ r.str.inf reads "the satisfaction of y is more sensitive to the satisfaction of x_i than to the satisfaction of x_j".

Language Services

- s.InfStrength: Yes, if $((x_i, y), (x_j, y)) \in$ r.str.inf is in M.

Example 5.6. Let L.Schedar be a language made by adding f.str.inf to L.Ankaa. The language is defined as follows.

Language

Schedar

Language Modules

F, r.inf.pos, r.inf.neg, f.map.abrel.g, r.str.inf

Domain

Set F of Fragments, r.inf.pos $\subseteq$ F $\times$ F, r.inf.neg $\subseteq$ F $\times$ F, and

$$\text{r.str.inf} \subseteq (\text{r.inf.pos} \times \text{r.inf.pos}) \cup (\text{r.inf.neg} \times \text{r.inf.neg}).$$

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{aligned} \alpha &::= x \mid y \mid z \mid \ldots \\ \beta &::= (\alpha, \alpha) \\ \gamma &::= (\beta, \beta) \\ \phi &::= \alpha \mid \beta \mid \gamma \end{aligned}$$

Mapping

$\mathscr{D}(\alpha) \in F$, $\mathscr{D}(\beta) \in$ r.inf.pos $\cup$ r.inf.neg, and $\mathscr{D}(\gamma) \in$ r.str.inf.

Language Services

s.PosInfluence, s.NegInfluence, s.InfStrength.

Figure 4 is a visualisation of a model made in L.Schedar. The figure shows that the satisfaction of AddRepEm is more sensitive to the satisfaction of IdIncLoc than it is to all other Fragments, whose satisfaction influences that of AddRepEm. •

The relation r.str.inf gives no indication about how to evaluate the relative strength of influence. Strength can be a function of covariance, for example. You then need to guess covariance values (in case you have no say about how exactly the satisfaction of x_i influences that of y) or to decide these values (when you can choose exactly how the satisfaction of x_i influences that of y). Both discussions are specific to the concrete RP instance that you are solving. For the former case, multivariate statistics [99, 124] provides general guidelines for estimating covariance. For the latter case, another discipline may provide relevant suggestions, and the discipline in question depends on what the Fragments are about. For example, if $x_1, \ldots, x_n$ reflect decisions on the architecture of an information system and y is a requirement about the scalability of that information system, then research on software architecture [113] is relevant.

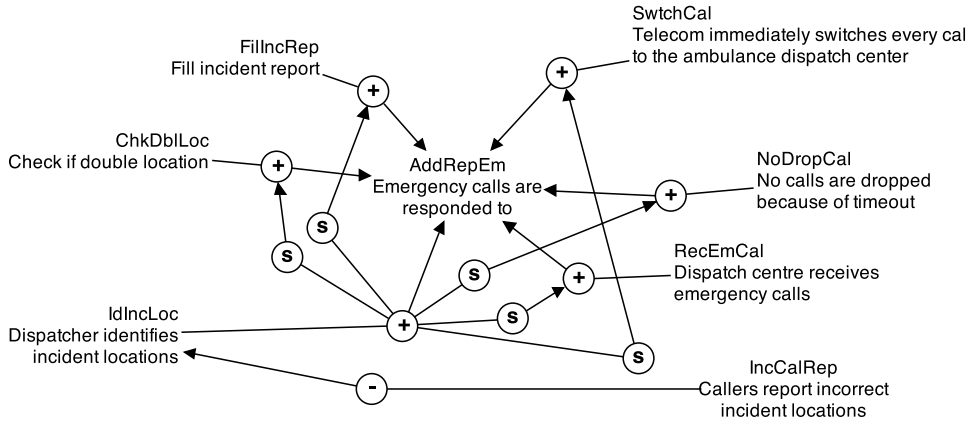


Figure 4: A visualisation of a model in L.Schedar.

You can see the absence of precise instructions in r.str.inf as a deficiency. However, this simply reflects the fact that it is in many cases required to call upon experts, among stakeholders or elsewhere, in order to produce relevant models. A language which uses that relation, rough as it is, is only pointing in the direction of relevant areas of expertise, rather than attempting to include some of the knowledge from them.

5.3.4 Summary on Influence Relations

The purpose of influence relations defined above is to represent that the satisfaction of some Fragments depends on the satisfaction of others. I defined several influence relations and a function which illustrated how to assign relative strength of influence to instances of positive and negative influence relations.

None of the influence relations came with with predefined levels of satisfaction. I said how to read the satisfied and not satisfied values, when there are only two levels of satisfaction, but I said nothing about cases when there are many levels of satisfaction.

This was acceptable precisely because influence relations are used when we have partial knowledge, due to discovery or indecision about how exactly to compute the satisfaction value $SatVal(y)$ of a Fragment y.

The story was that, as my knowledge about $SatVal(y)$ increases, I will want to stop using r.Influence, and want to use r.Influence[d] instances. As it further increases, I will want to use f.str.inf to indicate the relative strength of influence. If I knew even more, I could formulate a concrete function $SatVal(y) = f(SatVal(x_1),\ldots,SatVal(x_n))$, which I might revise at later iterations in problem-solving.

When you can formulate $SatVal(y) = f(SatVal(x_1),\ldots,SatVal(x_n))$, you have reached a point in problem-solving when influence relations alone represent less than you know about the influence of $x_1,\ldots,x_n$ on the satisfaction of y. At that point, you need a language with more complicated satisfaction scales, and functions assigning those values. I will return to this in Section 9.

5.4 How to Show the Rationale for Model Content?

A recurring concern in RE is to make justified models. A model is justified if the rationale for its content is acceptable to everyone involved in making and using that model (or at least to those having the authority to complain about the content of a model). The rationale explains why something is in the model. If the content of a model is contested, and nothing is given to settle the debate, then the model is not justified. If it is not justified, it is unclear whether the problem and solution it may represent are relevant at all.

Exercise 9: Show, in a model, information which justifies the content of that model

Define relations which can be used to show, in models, that some Fragments are arguments, reasons for having other parts of the model, such as other Fragments or relation instances. Can you do this with

one relation? If not, then why? How would you use these relations to determine if a model part is justified or not?

Checking if a model is justified can be done once it is completed. Another approach is to check every change of the model, to make sure that the change itself is justified. In both cases, the idea is that there are some properties that the model should have, and which must be satisfied in order to say that the model is justified. These ideas about justification are closely related to a central notion in program refinement.

Program refinement [137, 38, 68, 37] consists of replacing a piece of abstract program with a piece of more concrete program, the benefit being to delay lower-level detail to later steps of program development. This is related to the idea of incrementally adding detail discussed earlier, but I want to focus on another important idea in program refinement.

A central notion in program refinement are *proof obligations*. They are properties for which it is necessary to produce a formal proof, in order to claim that a particular program refinement is correct. The more concrete program a refines a more abstract program b if and only if all the specific proof obligations for that refinement relation are satisfied. In other words, you can say that there is a program refinement relation from a more concrete program a to a less concrete b if and only if all proof obligations are satisfied.

All relations defined so far in this tutorial come with conditions that must be satisfied by model elements, in order to have a relation instance between them. These appear in the slots of the corresponding Language Modules. For example, the "Reading" slot for r.ifm says that $(x, y) \in$ r.ifm reads that x adds details to y, and thus, that this relation instance should be in a model if the given informal condition is satisfied, namely, that x does add details to y.

The issue is that these conditions are not equally precise and unambiguous for all relations, and from there, not equally convincing to all those making and using models. Proof obligations remove, or at least reduce the need to debate whether a program a refines a program b: if proof obligations are satisfied, then it does, and anyone using the model can check for themselves if they are satisfied.

However, if I write $(x, y) \in$ r.ifm in a model, then my justification for the existence of that relation instance is, just as the definition of r.ifm says, my own judgment that x adds details to y. This might be fine if I am the only person using that model. But you cannot know from that model and its language *why* I concluded that x adds details to y. And this is a practical problem, because if you wanted to know, you would need to ask me, and that would take time and other resources away from more relevant uses.

As should be clear by now, problem-solving in RE involves working with partial information. So it is often simply not feasible to provide conditions as clearly verifiable as proof obligations.[6]

5.4.1 Support and Defeat

Justification can perform a similar role to proof obligations when information is partial or otherwise deficient. Justification consists of recording reasons for and against the inclusion of Fragments and relations in a model, and checking which of these are "accepted". I will consider "accepted" and "justified" to be synonyms. Reasons may come from model users, other stakeholders, or from anyone else who gives them. Justification comes with rules which define when something is "accepted".

To do justification, I will use a pair of relations called r.Support and r.Defeat. With them, I will be able to record arguments for and against parts of models. They will be used to deliver the following Language Services.

Language Service
DoesSupport: Does accepting x support accepting y as well in M?

Language Service
DoesDefeat: Does accepting x support rejecting (not accepting) y in M?

r.Support and r.Defeat also make it possible to define languages that can deliver such Language Services as, for example, "Why is it that x adds details to y?", "Why

[6] There are at least two reasons for this. One is that I may not know a clear enough and complete set of conditions to satisfy, for a relation instance to be present. This makes it less relevant to use a formal language, such as a formal logic, to define proof obligations. The issue is not that I cannot formalise something because the formalism is limited in some way, but that I do not know what exactly to formalise. So just as I have partial information about the problem to solve, I also have partial information about the problem-solving method that I am applying. Another reason is that partial information may change quickly. For example, stakeholders may say something at a meeting one day, and change their mind at the next. In such cases, formalisation may be left for later phases of problem-solving, and be restricted only to problem and solution information which is considered as more stable. For example, it may involve formalising some aspects of a system design which the stakeholders approved (more on this in Section 9).

is it that x influences y positively?", "Do stakeholders agree that x influences y positively?", and similar. The relations are defined as follows.

Relation

Support (r.sup)

Domain & Dimension

r.sup $\subseteq X \times X$, where X is either a set of Fragments or relation instances.

Properties

irreflexive, antisymmetric, and transitive.

Reading

$(x, y) \in$ r.Support reads "if x is accepted, then y should be".

Language Services

- s.DoesSupport: Yes, if there is $(x, y) \in$ r.Support in M.

In contrast to r.sup, r.def is intransitive. This is an important property, and comes from the idea that if x defeats y and y defeats z, then it cannot be that x defeats z. By defeating y, x removes the argument against z, and thereby is not defeating z.

Relation

Defeat (r.def)

Domain & Dimension

r.def $\subseteq X \times X$, where X is either a set of Fragments or relation instances.

Properties

irreflexive, antisymmetric, and intransitive.

Reading

$(x, y) \in$ r.Defeat reads "if x is accepted, then y should not be".

Language Services

- s.DoesDefeat: Yes, if there is $(x, y) \in$ r.Defeat in M.

The following example illustrates how to use r.sup and r.def to give reasons for and against instances of the r.ifm in a model.

Example 5.7. How would you define a language which should represent the incremental adding of detail to models, and reasons for and against the additional details that are added?

Let L.Diphda be a new language that can represent r.ifm instances, and Fragments as reasons for and against these instances. Moreover, it can be used to say that one Fragment is an argument for, or against another Fragment.

Language

Diphda

Language Modules

F, r.ifm, r.sup, r.def, f.map.abrel.g

Domain

F is a set of Fragments. r.ifm is over Fragments, so r.ifm $\subseteq$ F $\times$ F. A Fragment can act as a reason, or argument in favour or against a r.ifm instance or another Fragment, so that

$$\begin{aligned} \text{r.sup} &\subseteq (\text{F} \times \text{r.ifm}) \cup (\text{F} \times \text{F}), \\ \text{r.def} &\subseteq (\text{F} \times \text{r.ifm}) \cup (\text{F} \times \text{F}). \end{aligned}$$

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & (\alpha, \alpha) \\ \gamma & ::= & (\alpha, \beta) \\ \phi & ::= & \alpha \mid \beta \mid \gamma \end{array}$$

Mapping

α symbols denote Fragments, so $\mathscr{D}(\alpha) \in$ F. β symbols denote r.ifm instances, or instances of r.sup or r.def between Fragments, that is,

$$\mathscr{D}(\beta) \in \text{r.inf} \cup \text{r.sup} \cup \text{r.def}.$$

γ symbols denote r.sup or r.def from a Fragment to a r.ifm instance,

$$\mathscr{D}(\gamma) \in \text{r.sup} \cup \text{r.def}.$$

Language Services

s.DoesSupport, s.DoesDefeat.

A way to see L.Diphda, is that I take a model of L.Alpheratz as a basic model in L.Diphda, and then add arguments in favour or against r.ifm relation instances in the L.Alpheratz model.

In Example 5.1, Fragments RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, and FillIncRep described parts of what needs to be done, in order to satisfy AddRepEm. In Example 5.2, I added instances of r.ifm over these Fragments. A reason why I added these relation instances is that each of RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, and FillIncRep said *how* to satisfy AddRepEm. Moreover, SwtchCal says that the telecom switches the call, so that it says *who* is involved in satisfying AddRepEm. Similarly, RecEmCal and IdIncLoc also identified other positions, respectively, the dispatch centre and dispatcher, who have responsibilities in satisfying AddRepEm.

This leads to the following new Fragments that justify the said r.ifm instances:

SwtchCal says (is part of the answer to) how to satisfy AddRepEm. (HowSwtchCal)

SwtchCal says who is involved in satisfying AddRepEm. (WhoSwtchCal)

SwtchCal says when some events happen when satisfying AddRepEm. (WhenSwtchCal)

NoDropCal says how to satisfy AddRepEm. (HowNoDropCal)

RecEmCal says how to satisfy AddRepEm. (HowRecEmCal)

RecEmCal says who is involved in satisfying AddRepEm. (WhoRecEmCal)

IdIncLoc says how to satisfy AddRepEm. (HowIdIncLoc)

IdIncLoc says who is involved in satisfying AddRepEm. (WhoIdIncLoc)

ChkDblLoc says how to satisfy AddRepEm. (HowChkDblLoc)

FillIncRep says how to satisfy AddRepEm. (HowFillIncRep)

All of the above give reasons in favour of the various r.ifm instances. The following are these instances of r.sup:

(HowSwtchCal, (SwtchCal, AddRepEm)), (WhoSwtchCal, (SwtchCal, AddRepEm)),
(WhenSwtchCal, (SwtchCal, AddRepEm)), (HowNoDropCal, (NoDropCal, AddRepEm)),
(HowRecEmCal, (RecEmCal, AddRepEm)), (WhoRecEmCal, (RecEmCal, AddRepEm)),
(HowIdIncLoc, (IdIncLoc, AddRepEm)), (WhoIdIncLoc, (IdIncLoc, AddRepEm)),
(HowChkDblLoc, (ChkDblLoc, AddRepEm)), (HowFillIncRep, (FillIncRep, AddRepEm)).

Figure 5 shows a visualisation of the resulting L.Diphda model. r.sup relation instances give arguments in favour or r.ifm relation instances. r.sup instances are shown as white circles labeled "A+", connected to the Fragment that is the argument, and to the relation instance which the argument supports. •

Example 5.7 illustrated how to give one or more arguments in favour of individual r.ifm instances. I did not, for example, give arguments for or against other arguments, yet this can be done. It follows that I can represent that x is an argument in favour of y, and that z is an argument in favour of x, and then, that w is reason against z. In general terms, it allows me to represent the outcome of *argumentation*, the adding of arguments, as chains of r.sup and r.def instances. The following example illustrates this.

Example 5.8. James and Jill are modelling requirements for CADS. James made the model discussed in Example 5.7, visualised in Figure 5. Jill disagrees that FillIncRep adds details to AddRepEm by answering how AddRepEm should be satisfied. The reason why Jill disagrees, is that FillIncRep is not in the scope of AddRepEm, and is an administrative matter, to be discussed separately from how to satisfy AddRepEm, that is, of how to respond to emergency calls. This can be recorded in the model by adding the Fragment RepFillOutScp.

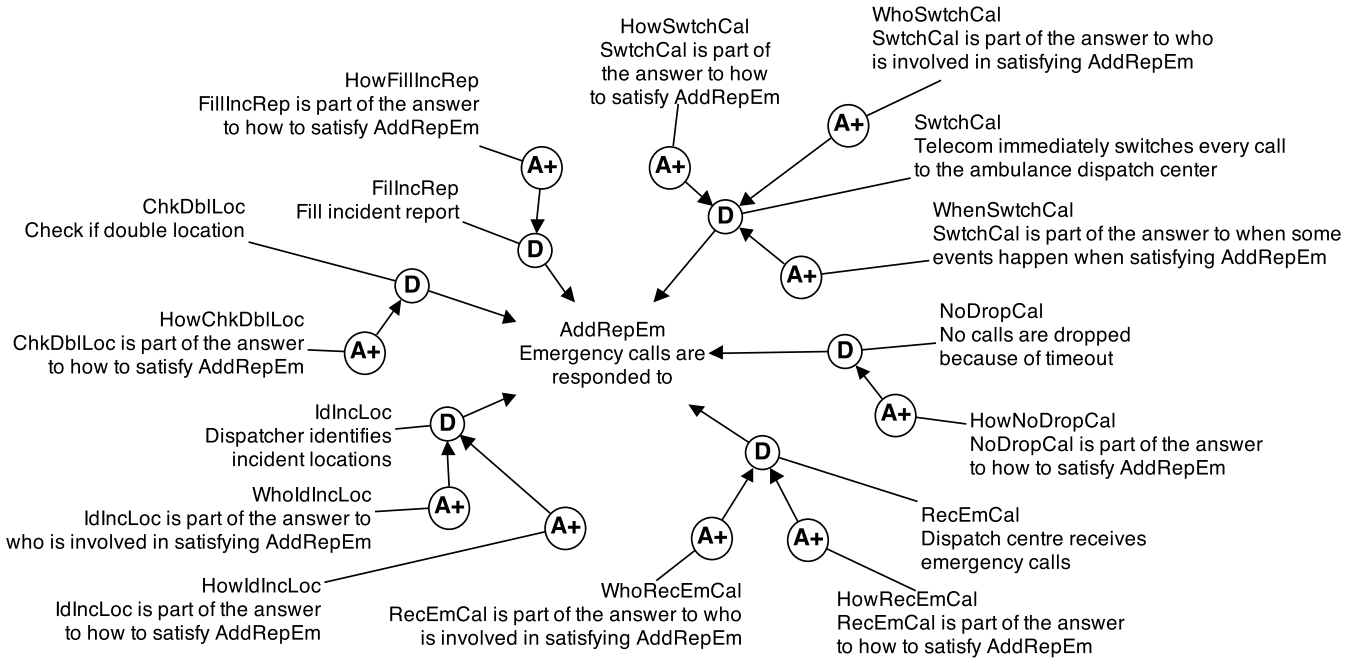


Figure 5: A visualisation of a model in L.Diphda.

To respond to an emergency call, it is not necessary to fill out an incident report. (RepFillOutScp)

And then, by adding the r.def instance

$$(\text{RepFillOutScp}, \text{HowFillIncRep}) \in \text{r.def}$$

This would result in updating Figure 5 by adding a Fragment node for RepFillOutScp and an r.def instance from it to HowFillIncRep. •

r.sup and r.def are similar in purpose to relations in existing languages in RE and elsewhere, which are used to represent the design rationale [85, 29, 87, 88, 106, 115, 114, 93, 81, 78, 75], that is, reasons for and against the content of models, or if we look at it from the perspective of the modelling process, then a record of why different model elements were added or removed.

The idea of representing design rationale with arguments for and against model elements is related to two important observations [110]. Firstly, many design and engineering problems are ill-defined, so-called *wicked problems*, lacking a clear scope and formulation, known optimal solutions, or known systematic processes for producing solutions. Secondly, solving such problems, therefore, cannot involve a known systematic process, but involves finding pieces of the problem and pieces of potential solutions, and collaboratively debating their pros and cons by giving arguments for and against these pieces, or their combinations. Such problem-solving ends rarely because one finds the best solution, but because of practical time and resource constraints. This makes it interesting to record the design rationale as arguments that led to modelling decisions, whereby the resulting models represent the problem and its solutions, together with explanations of why you were solving that problem instance and not another, and why you produced that or those solutions, and not others.

5.4.2 Accepted or Rejected

Having chains of r.Argue[p] instances raises the issue of *acceptability*. Acceptability is interesting, because something being acceptable is synonymous to it being justified. In Example 5.8, there was a chain made from $(\text{RepFillOutScp}, \text{HowFillIncRep}) \in \text{r.def}$ and

$$(\text{HowFillIncRep}, (\text{FillIncRep}, \text{AddRepEm})) \in \text{r.sup}$$

that is, HowFillIncRep was in favour of saying that FillIncRep adds details to AddRepEm, and then an argument against HowFillIncRep. Asking about acceptability in this case equates to asking this: Should $(\text{FillIncRep}, \text{AddRepEm}) \in \text{r.ifm}$ be used in problem-solving, given the said r.sup and r.def instances, or should it be ignored (do as-if it were not in the model at all)? So we need rules to compute acceptability.

Exercise 10: Define a procedure which computes if a model part is acceptable

> L.Diphda can represent arguments for and against in models. Given a model which includes arguments, which arguments are acceptable (justified), and which are not? How can you compute this? How can the ability to compute this be built into a language?

I will see acceptability as a value assigned to relata of r.sup and r.def instances.

There is a nuance to how to use that value in problem-solving. Instead of saying that acceptable elements should stay in a model, and unacceptable ones be removed, I will remove nothing from a model. (You can have different visualisations of the same model, some showing all, some only parts, so there really is no need to remove model parts.) Instead I will say that only acceptable elements should be used in problem-solving. The reason for this is that new elements and r.sup and r.def instances may change the acceptability of existing elements. This is because argumentation, in the form outlined above with a relation for supporting arguments and another for counterarguments, is a form of non-monotonic reasoning, a point made in philosophy, in relation to, for example, informal logic [134, 8, 67], and in artificial intelligence, in relation to argumentation systems [43, 25, 7] and defeasible logics [104, 117, 105].

While acceptability and satisfaction are values assigned to model elements, they are *different kinds of values, because they are used differently in problem-solving*. I said earlier that satisfying x amounted to doing successfully what x describes. If you think in terms of satisfaction, then *the acceptability value of x tells you if you should worry about the satisfaction of x at all.* If x is not acceptable, then it is irrelevant to problem-solving, and it does not matter, for example, how it influences other model elements. This makes it unnecessary to evaluate the satisfaction of x. If x is acceptable, then it makes sense to evaluate the consequences of satisfying or not satisfying it.

The following gives a rough idea about how to compute acceptability. Suppose that y supports x, and z defeats y, and that nothing else relates via argues relations to any of x, y, and z. What is the acceptability of x, y, and z? A common rule in argumentation systems in artificial intelligence [25] is that z is acceptable, since there is no argument against it. So because z is acceptable, and is an argument against y, then y is not acceptable (rejected). Finally, as y was in favour of x, and y is now not acceptable, then the convention is that x is rejected also, as the only argument in its favour is rejected. This is usually a bit more complicated, as there can be more than one arguments in favour and against any one element.

Example 5.9. To illustrate the computation of acceptability, I start with the simpler case, when a model has only one so-called "extension". An extension includes all

acceptable model parts. Depending on the language in which the model is made, and on the content of the model, it is possible to have models with more than one extension.

Figure 6(a) shows a visualisation of a L.Diphda model which takes the Fragments AddRepEm, FillIncRep, HowFillIncRep, and RepFillOutScp from earlier examples, and adds six new Fragments x1 to x6. The rationale relations matter for this example, not the specifics of actions or conditions these new Fragments describe.

Which Fragments in Figure 6(a) are acceptable? Consider first the leaves, and observe that there are no arguments against AddRepEm, FillIncRep, x2, x3, x5 and x6, so that they are acceptable. x6 supports x5. Since x5 is acceptable and is against x4, x4 is not acceptable. Consequently, it does not matter for the acceptability of x1 that x4 is against x1.

However, x3 is acceptable and attacks x1. I therefore need to choose if arguments against or arguments for are stronger, since this determines whether x1 is acceptable (as x5 is an acceptable argument in its favour). I take the cautious approach, and decide that negative arguments cancel positive ones, and therefore, x1 is not acceptable. It follows that RepFillOutScp is acceptable, and HowFillIncRep is not. So HowFillIncRep is no longer an acceptable argument in favour of the r.ifm relation from FillIncRep to AddRepEm. This leads me to a second decision, which is whether the absence of a positive argument in favour of a model part, also means that that model part is not acceptable. I will assume that it is acceptable, as I did the same for, for example, x6 which also lacks positive arguments in its favour.

The resulting acceptability values are shown as additional markers on model elements in Figure 6(b). The model there has exactly one extension, and it includes all model parts which are marked with the acceptability value 1.

Figure 6(c) shows what happens when there is an additional r.def instance, which leads to two extensions. For a designer of the language, the possibility for alternative extensions means that the language could suggest which of the extensions to choose.
•

I use Dung's definition of acceptability [43]. This is convenient because it is simple and generalises many others in artificial intelligence. The rough idea is similar (but not the same, as explained in Example 5.10) to that explained above with x, y, and z and in Example 5.9. The main difference is that in his graphs, all edges are instances of the so-called "attack" relation. Attack corresponds to my r.def, but there are no relations in to capture supporting arguments. This is not a major issue, but will influence how I convert my models into his. I will call his models "argumentation frameworks".

I need a function that delivers the following Language Service.

Language Service

IsAcceptable: Is w acceptable in W, given relations r.sup and r.def over W?

The function f.acc below takes instances of r.sup and r.def over some set W, and determines if some $w \in W$ is acceptable or not.

Function

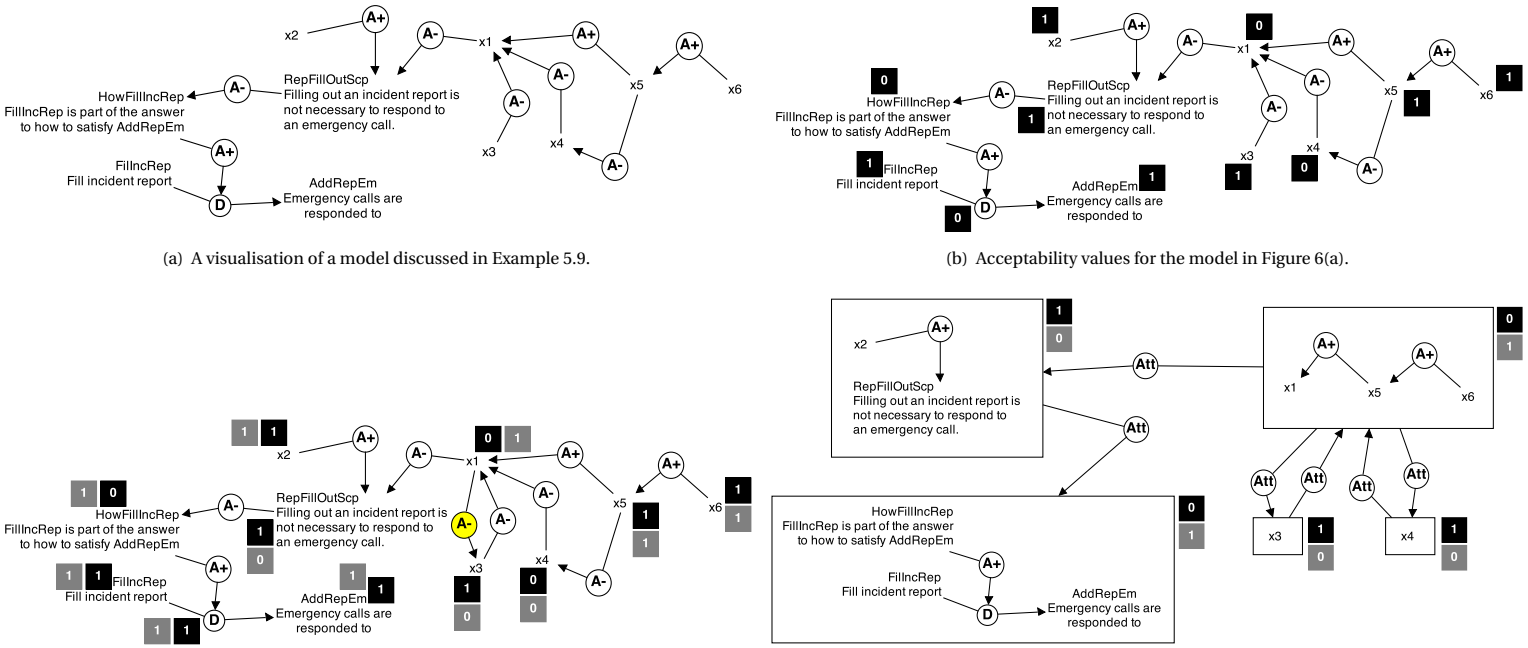
Accepted
(f.acc)

Input

A Fragment or relation instance w, a set W such that $w \in W$, $A_+ \subseteq$ r.sup, and $A_- \subseteq$ r.def, where r.sup $\subseteq W \times W$ and r.def $\subseteq W \times W$.

Do

1. Let G(W, r.sup) and G(W, r.neg) be graphs made with f.map.abrel.g.
2. Let G(w, W, r.sup) be the subgraph of G(W, r.sup), which includes only the paths of G(W, r.sup) which end in w.
3. Let G(w, W, r.neg) be the subgraph of G(W, r.neg), which includes only the paths of G(W, r.neg) which end in w.
4. Let C include all connected components of G(w, W, r.sup).
5. Let K include every node from G(w, W, r.sup), which is not in a connected component in C.
6. Make an empty set, call it *Arg*, and let l_{Arg} be a function which will return the label of each element in *Arg*.
7. For each $c \in C$, add a to *Arg* and let $l_{Arg}(a) = c$.



(a) A visualisation of a model discussed in Example 5.9.

(b) Acceptability values for the model in Figure 6(a).

(c) There are now two extensions.

(d) Dung argumentation framework made by applying f.acc to the model in Figure 6(c).

Figure 6: Illustration of how to compute acceptability values.

8. For each element $k \in K$, add a to Arg and let $l_{Arg}(a) = k$.
9. Make the graph $AF = (Arg, Att)$, with $Att \subseteq Arg \times Arg$ and let Att be empty.
10. For each $(w_i, w_j) \in$ r.def in G(w, W, r.def), add an edge $(a_i, a_j) \in Att$ to AF, so that a_i is such that, either
    - $l_{Arg}(a_i) = w_i$, if $w_i \in K$, or
    - $l_{Arg}(a_i) = c_i$, if $c_i \in C$ if w_i is a node in the connected component c_i,

    and a_j is such that, either
    - $l_{Arg}(a_j) = w_j$, if $w_j \in K$, or
    - $l_{Arg}(a_j) = c_j$, if $c_j \in C$ if w_j is a node in the connected component c_j.
11. The graph $AF = (Arg, Att)$ is a Dung argumentation framework.
12. Use an existing algorithm [96] to compute the acceptability of arguments in AF.
13. If an argument a in AF is acceptable, $l_{Arg}(a) = k$ and $k \in K$, then that element in W is acceptable.
14. If an argument a in AF is acceptable, $l_{Arg}(a) = c$ and $c \in C$, then all elements of W which are in c are acceptable.
15. Let $Acc(W)$ include all acceptable elements of W.

Output

The set $Acc(W)$.

Language Services

- s.IsAcceptable: Yes, if w is in the set $Acc(W)$.

Example 5.10. To clarify how f.acc works, recall that a Dung argumentation framework $AF = (Arg, Att)$ is a graph where nodes represent arguments and edges the attack relations. If an argument attacks another, then believing in the former tells us that we should not believe in the latter, or that the former is evidence against the latter. So the attack relation equates in use to r.def. But there is no relation in an argumentation framework which corresponds to r.sup. I therefore have to decide what we do with r.sup when making a Dung argumentation framework. f.acc shows one way to do this.

Applying f.acc to the model in Figure 6(c) gives the argumentation framework visualised in Figure 6(d). Tha figure also shows the acceptability values in two extensions of the framework. Note the differences between the extensions in the Dung argumentation framework and the extensions in Figure 6(c). They are due to the choice, in f.acc, to equate a Dung argument to a connected component over r.sup instances. •

There are algorithms to find connected components of a graph [69] and to compute extensions of Dung argumentation frameworks [96]. All nodes in a Dung argumentation framework (called arguments there) are considered as acceptable if they are in an extension of the given argumentation framework.[7]

Asking that a relation instance x in a model is acceptable according to f.acc can be seen as an analogue to a single proof obligation, in the sense that it is a single condition that the relation instance needs to satisfy in order to be relevant for problem-solving.[8] In contrast to proof obligations, which can depend on the properties of x and so be specific to the type of x, acceptability is independent from the properties of x and therefore, it can apply to any x, in any model, in any modelling language. For example, if x is a relation instance, proof obligations may be sensitive to x being reflexive or not, symmetric or not, and so on, while acceptability of x depends solely on those concrete reasons for and against x that we have in a particular model (not a modelling language, and so not *any* model, but exactly *that* model). The benefit is that we can build acceptability into a language when we lack a clear idea for proof obligations. The limitation is precisely that it is independent from the properties of x and so involves collecting and confronting anew reasons for and against.

[7] I leave it to the reader to look up the types of extensions, how they differ, and what consequences using one or another type of extension in f.acc would have [43, 116].

[8] It is an analogue, because it is a justification and not a deductive proof, as in a formal logic with a monotonic syntactic consequence relation. Namely, if you have a deductive proof of some x in a monotonic logic, then you can still prove x regardless of any new formulas that you are adding, while having a justification for x is sensitive to new formulas, in that new formulas can block proofs which we previously had. As Pollock observes, justification is defeasible reasoning [104]: "[...] inductive reasoning is not deductive, and in perception, when one judges the colour of something on the basis of how it looks to him, he is not reasoning deductively. Such reasoning is *defeasible*, in the sense that the premises taken by themselves may justify us in accepting the conclusion, but when additional information is added, that conclusion may no longer be justified. For example, something's looking red to me may justify me in believing that it is red, but if I subsequently learn that the object is illuminated by red lights and I know that that can make things look red when they are not, then I cease to be justified in believing that the object is red."

5.5 How to Combine Relations?

Suppose you have a language that can represent r.ifm and r.inf instances over Fragments, and that it lets you have two relation instances between same Fragments. For example, you could have a model with $(x, y) \in$ r.ifm and $(x, y) \in$ r.inf. First of all, would you want the language to allow this in models? And if you do, then, does knowing that x both influences and informs y tell you something more than what these two relation instances tell you each on its own? When it does tell you more, then I will say that the relations interact.

When a language has more than a single relation, the challenge is to decide if these relations interact or not, and if the they do, then how to use their interactions.

If relations interact, then it matters for instances of a relation r.A that there exist instances in the model of another relation r.B. Section 5.5.1 focuses on the simpler case of independence, and Section 5.5.2 on interaction.

5.5.1 Independent Relations

L.Diphda included three relations and they were not interacting. It is a permissive language, as it imposes no constraints at all on how the presence of some relation between two Fragments x and y influences the presence or direction of other relation instances between the same pair of nodes. In other words, the definition of the language is silent on how, if in any way, the relations in it are interacting.

This is unlikely to cause problems if its models are such that there is only one relation instance over any two Fragments. When there are two or more edges between two nodes, then it may be unclear how to read this combination of relation instances. If there are two nodes, x and y, such that $(x, y) \in$ r.ifm and $(x, y) \in$ r.def, then what can you conclude about these two nodes? The language itself does not say if this is a modelling error, or is somehow useful in a model.

5.5.2 Interacting Relations

The problem with fitting different relations together in a language, and especially if the relations are only informally defined, is that it may allow models that convey unintended information to their users. There is no guarantee that all unintended information will be benign in problem-solving, so we are obliged to worry about how relations interact and to sanction problematic interactions.

I will use L.Achernar below to illustrate this discussion. It has the inform relation and the positive and negative influence relations.

Language

Achernar

Language Modules

F, r.ifm, r.inf.pos, r.inf.neg, f.map.abrel.g

Domain

Set F of Fragments, r.ifm $\subseteq F \times F$, r.inf.pos $\subseteq F \times F$, and r.inf.neg $\subseteq F \times F$.

Syntax

Same as L.Alpheratz.

Mapping

$\mathscr{D}(\alpha) \in F$ and $\mathscr{D}(\beta) \in$ r.ifm $\cup$ r.inf.pos $\cup$ r.inf.neg.

Language Services

Same as r.ifm, r.inf.pos, and r.inf.neg.

L.Achernar simply puts together several relations, while still making sure that the language deliver all the Language Services that the relations separately could. But, it will be clear below that the modeller has to invest significant effort with this language in order to make unambiguous models. One reason for this is that the language definition does not say how relations interact.

For example, suppose that a L.Achernar model includes, among others, the Fragments x and y and the following two relation instances.

$$(x, y) \in \text{r.inf.pos}$$
$$(x, y) \in \text{r.inf.neg}$$

Does, then, x influence positively or negatively y? The answer is not *in* the definitions of L.Achernar and of the influence relations, as they say nothing about such cases. It is also irrelevant to look *outside* these definitions, since I they are neither equivalent, nor subtypes of others that are defined outside this tutorial. The only remaining option is that the influence relations, and therefore the L.Achernar language, leave it up to the model user to decide for herself if x positively or negatively influences y.

Exercise 11: Define rules for how r.inf.pos, r.inf.neg, and r.ifm interact

Suppose a model can represent positive and negative influence, and inform relations between Fragments. Consider all possible combinations of relation instances between two Fragments, and define rules for what to do in each case. How would you define these rules in a language? How would you define a new language from L.Achernar which includes these rules?

If the language definition does not explain what to do with relation interactions, then the language does not provide support to its users, on how to deal with these combinations. The language can include Language Services focused on interactions, such as the following.

Language Service

NegWins: If $(x, y) \in$ r.inf.pos and $(x, y) \in$ r.inf.neg, then does x influence positively or negatively y?

Suppose that the answer is: x influences y negatively, and remove $(x, y) \in$ r.inf.pos. This answer can be added to a language as a function, for example, to L.Achernar. The new language would deliver s.NegWins.

The more general point is that once there is more than one relation in a language, it is useful to explain how to use each possible interaction between these relations. This may simply result in explicitly stating in the language definition that it is up to the modellers to decide what to do with interactions.

Consider now all possible interactions of relations in L.Achernar. For each interaction, I give a rule which could be applied.

1. $(x, y) \in$ r.ifm and $(x, y) \in$ r.inf.pos is allowed, and indicates that x informs y, and in such a way that satisfying it positively influences the satisfaction of y.

2. $(y, x) \in$ r.ifm and $(x, y) \in$ r.inf.pos can be handled in different ways, of which two are below:

    - One option is to decide that is not allowed, and one of the two should be removed from the model. This can be motivated as follows: if y is adding details to x, this is because it is clearer how to satisfy y and less clear how to satisfy x, so that I will not be looking to satisfy y by satisfying x. (If the language had the relations for justification, then it would not be necessary to remove one of the two relation instances from the model. It would be enough to make one of the two unacceptable.)
    - Another option is to allow this if y adds such details to x by explaining the consequences which will occur if x is not satisfied, so that if x is satisfied, these consequences will occur, which is captured by the positive influence relation.

3. $(x, y) \in$ r.ifm and $(y, x) \in$ r.inf.pos should be handled in the same way as the case $(y, x) \in$ r.ifm and $(x, y) \in$ r.inf.pos.

4. $(x, y) \in$ r.ifm and $(x, y) \in$ r.inf.neg can be handled via analogous options to those for $(y, x) \in$ r.ifm and $(x, y) \in$ r.inf.pos, except that there is negative influence.

5. $(y, x) \in$ r.ifm and $(x, y) \in$ r.inf.neg should be handled in the same way as the case $(x, y) \in$ r.ifm and $(x, y) \in$ r.inf.neg.

6. $(x, y) \in$ r.inf.pos and $(x, y) \in$ r.inf.neg is not allowed, and one of the two should be removed.

7. $(y, x) \in$ r.inf.pos] and $(x, y) \in$ r.inf.neg can be handled in different ways, and two are below for illustration:

    - Remove one of the two influence relations.
    - Consider that these two influence relations represent a feedback mechanism, and leave them in the model.

The discussion above leads to three important remarks. The first is about incompleteness in language definition, the second on how completing a language definition suggests new Language Services, and the third on how to define new relations from combinations of existing ones.

- The discussion of relation interactions shows that the definition of L.Achernar was incomplete. It is necessary to consider each of the possible interactions, check if the language definition says something about them, and *if not, then decide what to do with the interaction, that is, make new language design decisions.* So I decided that when x both positively and negatively influences y, one of these two influence relations should be removed.

- The second important remark is that looking at all possible interactions suggests new Language Services. For example, adding these new rules for interactions to the language can answer, for example, "Is a model M in L.Achernar correct?". A model in L.Achernar was correct as long as the model did not violate the actual

definitions of the individual relations (for example, it could violate them if it had two positive influence relations between same two nodes). If the rules on interactions are added to the language, then model correctness gets a new definition in it.

- A particular case of interaction, or more of them, can be used to define new relations in a language. For example, I can define a new relation called r.Feedback[mixed] as a binary relation that exists between Fragments x and y if and only if there are $(x, y) \in$ r.Influence[positive] and $(y, x) \in$ r.Influence[negative]. This new relation is not a primitive of the language, as it is equivalent to a particular pattern of instances of other relations in the language.

5.6 Summary on Relations

The following are the main ideas from the preceding sections on relations:

- When defining a relation, it is useful to say, at least, what it relates, what to do to add its instances to models, and its formal properties (which are necessary if you want to do computations over graphs induced by the relation instances.

- The influence relations illustrated how you can have relations that reflect differences in how much you know when making a model. For example, if you think there is influence of satisfying x on satisfying y, and you do not if that influence is positive or negative, or how strong it may be relative to others that influence the satisfaction of y, then you can use r.inf. If you then decide or discover that the influence is positive, then you can represent this with an instance of r.inf.pos.

- Rules for the use of a relation are central to its definition, as they give the conditions to satisfy, in order to add a relation instance to a model. Ideally, use rules should be such that any model user can check if a relation instance is correct with regards to its use rules, that is, if it satisfies the required conditions. When you have use rules that are difficult to verify, you can augment them with a justification process, which was illustrated with f.Accepted.

- When a language has two or more relations, and when instances of different relations can be between the same model elements, then it is necessary to consider all possible relation interactions, decide how to read and use them, and how to capture these instructions in the language definition.

There are many other topics on defining relations in RMLs, and some of them will be discussed in the next sections. Section 6 focuses on how guidelines for modelling can be added to language definitions, but shows also how guidelines can suggest new relations and appear in the definitions of these relations. Section 7 introduces categories, and illustrates how relations can be restricted to specific Fragment categories, which can reduce the number of relation interactions. Section **??** looks at how to produce proofs of satisfaction from models, which is required to solve DRP instances, and shows one way of mapping relation instances to formulae in a formal logic. Section 8 uses n-ary relations in order to represent alternatives in models.

6 Guidelines

Overview and Motivation

This section is on how to define *guidelines* for problem-solving in RMLs. Guidelines recommend how to do something in problem-solving, so as to move closer to a solution. The section looks at the following questions.

1. How to find guidelines for problem-solving, and embed them in RMLs? (Section 6.1)
2. How to combine guidelines into new, more complicated ones? (Section 6.2)
3. How to strengthen or weaken guidelines, and why? (Section 6.3)

Guidelines suggest how to do problem-solving in RE. They may recommend how to elicit requirements, how to make them more precise, how to prioritise them, how to validate them with stakeholders, and so on.

Guidelines have a narrow scope when they focus on a specific task in problem-solving. An example is f.Accepted. Guidelines that have broader scope address complicated problem-solving tasks. Suppose, for example, that you know the following rough recommendation:

> Add details to the model until all stakeholders have agreed that the most detailed elements are detailed enough.

To help you apply this recommendation, a language clearly needs r.ifm, so that you can represent the adding of detail and identify the most detailed model elements. The language also needs to enable stakeholders to express agreement and disagreement, to represent reasons for agreeing or disagreeing, and to help you identify what the stakeholders agree and disagree on. You can do the first two with r.sup and r.def, and if you say that any acceptable model element is also agreed upon, then the language can use f.acc to find what is agreed and disagreed on.

RMLs and guidelines are intertwined, in that it is difficult to design one while ignoring the other. If a language should help us address an issue during problem-solving, then it will be designed to fit ideas and experience of how such issues should be addressed. An Language Service summarises the issue to address, guidelines tell you what to do to address the issue, and the language should deliver the Language Service.

For example, the inclusion of a relation in a language reflects decisions about what the language should help its users with, that is, which Language Services it should deliver. A guideline may suggest that you should first add details to model elements, and then look for, for example, how the satisfaction of some influences that of others. To apply the guideline, you need a language that can represent the increase in details in model elements, and how the satisfaction of some influences that of others.

Sections 6.1 and 6.2 illustrate how to go from identifying an issue, to guidelines for addressing it, and to new Language Services and Language Modules which help apply these guidelines and embed them in language definitions. Section 6.3 illustrates the ideas of strengthening and weakening guidelines and why that may be relevant.

6.1 How to Find Guidelines in Recurring Arguments?

L.Alpheratz can be used to represent that some Fragments add details to others. But it did not suggest how to find new Fragments which inform existing ones. It could not deliver the following Language Services:

- **s.HowToAddDetails**: Given a Fragment x, how to find a new Fragment y which adds details to x, that is, such that $(y, x) \in$ r.ifm?
- **s.WhyAddsDetails**: Given two Fragments x and y such that $(y, x) \in$ r.ifm, why does y add details to x?

More detailed Fragments can be found, for example, through further elicitation, analysis of comparable RP instances, by drawing on experience with comparable systems and in related domains, and so on.

Exercise 12: Define a language which delivers s.HowToAddDetails and s.WhyAddsDetails

> What does a language need to deliver s.HowToAddDetails and s.WhyAddsDetails? Does it need new relations, new functions, or otherwise? How are these new relations or functions related to r.ifm?

If I want guidelines that are independent from the specific domain or RP class, I can look at various existing models that represent the increase in details of Fragments. The aim is to find regularities in the differences between Fragments that are related by r.ifm instances.

Take Example 5.7. There are patterns in the arguments given for r.ifm instances. The arguments are similar in HowSwtchCal, HowNoDropCal, HowRecEmCal, HowIdIncLoc, HowChkDblLoc, and HowFillIncRep, in that they argue for the presence of r.ifm instances by saying each time, that a Fragment x adds details to Fragment y by indicating *how* actions or conditions that y describes are, respectively, executed and satisfied. There are also similarities in the rationale WhoSwtchCal, WhoRecEmCal, and WhoIdIncLoc, where the additional details always say something about *who* is involved in satisfying the conditions that the informing Fragment describes.

While looking for rationale patterns may not lead to universally applicable guidelines that are good for all languages, it can still help deliver additional Language Services relative to L.Alpheratz.

If I find recurring reasons for adding new Fragments, and you and I agree that they are sufficiently relevant and generic to build them into a language, then I can document parts of how you and I use the language into that language. The language embeds more of our conventions on its use. While this may result from our joint work on models, it also means that we will be recommending those ways for use to anyone interested in making models with that language. For example, if you use L.Alpheratz, then you also accept that the inform relation is irreflexive, antisymmetric, and transitive; otherwise, you are using another language, not L.Alpheratz.

Given some Fragments about the CADS in Example 5.7, I can ask several questions for any given Fragment x, including who does the action or satisfies the condition that x describes, how, when, where, and for whose benefit. If another fragment y answers at least one of these questions for the action or condition in x, then y is adding details to x. If you and I agree that asking such questions is relevant, we can define the following Language Module.

Function

Add details
(f.add.ifm)

Input

Fragment x.

Do

1. Ask the following questions about x:
   - *Who*: Who does (satisfies) x?
   - *How*: How is x done (satisfied)?
   - *When*: When is x done (satisfied)?
   - *Where*: Where is x done (satisfied)?
   - *WhoFor*: Who needs x to be done (satisfied)?

   Above, "does" is used if x describes actions; "satisfies" if it describes conditions.
2. Define sets F_q and R_q, for $q \in \{Who, How, When, Where, WhoFor\}$, such that:
   (a) each $y \in \mathsf{F}_q$ answers the question q for x,
   (b) if y answers the question q for x, then there is $(y, x) \in$ r.ifm in R_q.

Output

Sets F_q and R_q, for $q \in \{Who, How, When, Where, WhoFor\}$.

Language Services

- s.HowToAddDetails: Apply f.add.ifm to Fragments in a given model M, and add the resulting sets back to M.
- s.WhyAddsDetails: If R_q is output by applying f.AddsDetails, and $(y, x) \in R_q$, then y adds details to x because it answers the question q for x.

The function f.add.ifm suggest finding more detailed Fragments via five questions. All new Fragments go in the sets F_q. When a Fragment $y \in \mathsf{F}_q$ answers a question q for x, then I also add a relation instance $(y, x) \in$ r.ifm and it goes in R_q.

In order to keep the information in models, about which Fragment answers which questions, I define five new unary relations on instances of r.ifm. The idea is that, if $(y, x) \in$ r.ifm and y answers the question q for x, then there will be an instance of a unary relation r.q on $(y, x) \in$ r.ifm. The relations are defined with the following template, where $q \in \{Who, How, When, Where, WhoFor\}$.

Relation

Answers question q (r.q)

Domain & Dimension

r.q $\subseteq R$, where R is a set of r.ifm instances.

Properties

None.

Reading

$r \in$ r.q, where $r = (y, x)$, reads "y adds details to x by answering question q for x".

Language Services

- s.WhyAddsDetails: If $(y, x) \in$ r.q, then y adds details to x because it answers the question q for x.

Example 6.1. How would you define a language that has r.ifm and all five r.q relations? The language L.Hamal below has these relations.

Language

Hamal

Language Modules

F, r.ifm, r.Who, r.How, r.When, r.Where, r.WhoFor, f.add.ifm

Domain

F is a set of Fragments. r.ifm $\subseteq F \times F$, r.q $\in$ r.ifm, for every

$$q \in \{Who, How, When, Where, WhoFor\}$$

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & (\alpha, \alpha) \\ \gamma & ::= & Who \mid How \mid When \mid Where \mid WhoFor \\ \delta & ::= & \gamma.\beta \\ \phi & ::= & \alpha \mid \beta \mid \delta \end{array}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in F$. β are for r.ifm instances, that is, $\mathscr{D}(\beta) \in$ r.ifm. δ symbols denote r.q instances, $\mathscr{D}((Who.\beta)) \in$ r.Who, $\ldots, \mathscr{D}((WhoFor.\beta)) \in$ r.WhoFor.

Language Services

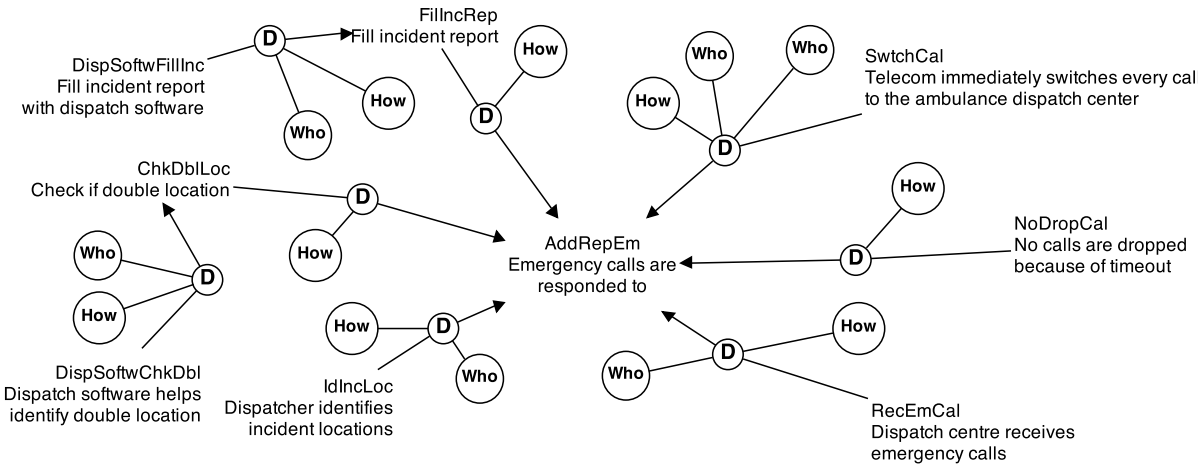
- s.WhyAddsDetails: If $q.(y, x) \in$ r.q, then y adds details to x because it answers the question q for x.

Figure 7 is a visualisation of a model in L.Hamal. It shows r.ifm instances and questions associated to each of these instances. The model was made by applying f.add.ifm to the Fragments in Example 5.1. •

6.2 How to Make Composite Guidelines?

The function f.add.ifm gave guidelines on how to add instances of one relation, r.ifm, with the side-effect that you added new relations, r.q over instances of r.ifm. The aim now is to define guidelines which rely on several relations and functions. As with f.add.ifm, the result will be a function.

Adding details to model elements, and then evaluating how the satisfaction of



[t]

Figure 7: A visualisation of a model in L.Hamal.

some influences that of others, are closely related to the issue of operationalisation in RE. The basic guideline in operationalisation can be stated as the rule *Op* below, and is inspired by analogous notions in *KAOS*, *Tropos*, and *Techne*.

> *Op*: Add details to model elements until the most detailed ones are judged as detailed enough that it is known how to satisfy them, and satisfying them results in satisfying all the least detailed model elements.

The guideline assumes that I start with Fragments that say what needs to be satisfied and, or executed, but that it is not clear how or who will do it. Operationalisation is the process by which I need to find and decide who and how make sure that these initial Fragments are satisfied.

Exercise 13: Define a function for operationalisation

> Define a function which checks if a Fragment in a model is operationalised, according to the rule *Op* in the text. Which relations do you need to have over Fragments in a model, in order to check if a Fragment is operationalised? Are there other guidelines which this operationalisation function uses?

To make a function inspired by the operationalisation guideline, you need r.ifm and r.inf.pos to represent, respectively, the increase in detail and the influence on satisfaction. You also need f.add.ifm to find new more detailed Fragments. Finally, you want this function to deliver the following Language Service.

Language Service

AreOpr: Are all Fragments in W operationalised?

The function is f.opr.all and is defined as follows.

Function

Operationalise all Fragments in a set
(f.opr.all)

Input

Set W of Fragments.

Do

1. Let X be an empty set, add all members of W to X.
2. Apply f.add.ifm to every Fragment $w \in X$, and add to X all new Fragments which you thereby find. If a Fragment $y \in X$ is detailed enough that it is known how to satisfy and, or execute what it describes, and it is known who takes the responsibility to do so, then do not apply f.add.ifm to y.
3. For every $(a,b) \in$ r.ifm, where $a,b \in X$, check if there should be $(a,b) \in$ r.inf.pos or $(a,b) \in$ r.inf.neg and if yes, then add it. Stop when it is known how the satisfaction of each more detailed Fragment influences the satisfaction of the Fragment to which adds details.
4. If there is a set $Z \subseteq X$ such that satisfying all Fragments in Z positively influences the satisfaction of all Fragments in W, and there are no Fragments in $W \setminus Z$ which inform those in Z, then stop. Otherwise, go back to step 1 above.

Output

Set Z of Fragments which are said to operationalise all Fragments in W.

Language Services

- s.AreOpr: Yes, if there is a set Z made by applying f.opr.all to W.

Example 6.2. f.opr.all can work with models of languages which have r.ifm, r.inf.pos, and r.inf.neg. L.Acamar below has these relations, and so can include f.opr.all.

Language

Acamar

Language Modules

F, r.ifm, r.inf.pos, r.inf.neg, f.add.ifm, f.opr.all

Domain

F is a set of Fragments, r.ifm $\subseteq$ F $\times$ F, r.inf.pos $\subseteq$ F $\times$ F, and r.inf.neg $\subseteq$ F $\times$ F.

Syntax

Same as L.Alpheratz.

Mapping

$\mathscr{D}(\alpha) \in$ F and $\mathscr{D}(\beta) \in$ r.ifm $\cup$ r.inf.pos $\cup$ r.inf.neg.

Language Services

Those of r.ifm, r.inf.pos, and r.inf.neg.

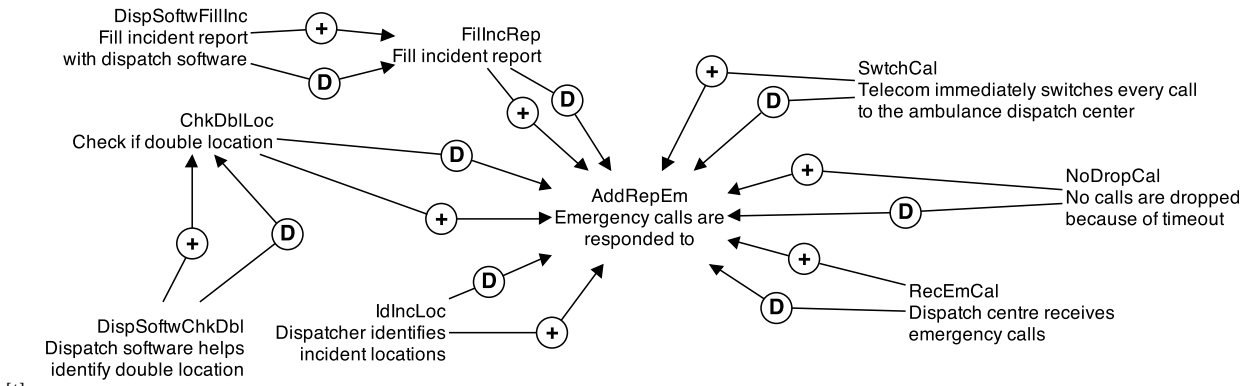
Figure 8 is a visualisation of a model in L.Acamar, made by applying f.opr.all to the Fragment AddRepEm. •

6.3 How to Strengthen or Weaken Guidelines?

f.opr.all uses f.add.ifm, and therefore, produces also graphs $G_{I[q]}$ for various questions q. f.opr.all also says that we should not apply f.add.ifm to those Fragments that are detailed enough, and a Fragment is, if it is known how to satisfy it and who is responsible for doing so. Notice, then, that f.opr.all did not define "being detailed enough" by the presence or absence of r.q relations, for, for example, *How* and *Who* questions. Instead, f.opr.all made no commitment about what exactly needs to be satisfied, in order for a Fragment to be "detailed enough".

If you want to define more precisely the conditions that a Fragment should satisfy, to be detailed enough, this can be done with another function. In that function, call it f.opr.all.b, all is identical to f.opr.all, except that the second step is replaced by the following:

> Apply f.add.ifm to every Fragment $w \in X$, and add to X all new Fragments which you thereby find. *A Fragment a is detailed enough if both Who and*



[t]

Figure 8: A visualisation of a model in L.Acamar.

How questions are answered for that Fragment, and do not apply f.add.ifm to that Fragment.

Above, the italics mark the part which differs relative to f.opr.all. The difference is that now use r.q in judging if a Fragment is detailed enough.

Verifying if a Fragment is detailed enough is simpler in f.opr.all.b than in f.opr.all, as it involves checking for the presence of r.Who and r.How instances, while in f.opr.all, you would have had to read the individual Fragments, to say if they are detailed enough.

While f.opr.all.b did make it easier to check if a Fragment is detailed enough, it did not necessarily result in a better guideline, since it is easy to find examples of Fragments which would be detailed enough for f.opr.all.b and not for f.opr.all. For example, answering a *Who* question does not necessarily identify who is responsible, only who is involved in satisfying what the Fragment describes. In short, the guideline documented in f.opr.all.b gives more precise and clearer instructions on what to do than f.opr.all, but neither function gives precise and clear sufficient conditions for a Fragment to be detailed enough.

Suppose that there are *new* conditions (which are neither in f.opr.all, nor f.opr.all.b) that a Fragment has to satisfy, in order to be considered detailed enough. Let f.opr.all.c be the function made by adding these new conditions to f.opr.all.b. For example, the new conditions are that a Fragment is detailed enough if and only if all q questions are answered for it. I will say that f.opr.all.c is *stronger* than f.opr.all.b, and that the former was made by *strengthening* the latter.

Strengthening a guideline involves adding conditions to check when applying the guideline, or to check in order to establish if the guideline is correctly applied. Weakening is the opposite, and consists of removing conditions that need to be checked.

As an additional illustration, remark that I said nothing about negative influences among Fragments. It follows that any of the three operationalisation functions can produce a set Z that operationalises its input set X, and we could have had negative influence relations between members of Z. One way to strengthen each of these functions is to add to each of them the condition that there can be no negative influences between members of Z.

6.4 Summary on Guidelines

The following are the main ideas discussed for guidelines:

- Guidelines recommend how to put the language to work when doing problem-solving. I can embed guidelines into the definition of the language, and in that way force specific ways of using it.

- You can define narrow guidelines on, for example, how to add a new relation instance to a model. In this tutorial, such guidelines appeared in use rules for relations. You can also combine narrow guidelines into broader ones, which use several relations, functions, or otherwise (other kinds of Language Modules introduced later in this tutorial), to deliver more complicated Language Services.

- Guidelines can be strengthened or weakened. I made no suggestions about universal rules on whether to strengthen or weaken a guideline. The stronger a guideline is, the more demanding it is on those involved in modelling, as there are more conditions to satisfy to use the language correctly. There may be situations in which this is not realistic, and consequently makes the language difficult to apply correctly, or makes it inapplicable.

- While experienced users of a language can suggest guidelines, it is also possible to identify guidelines by looking at recurring arguments for modelling decisions.

7 Categories

Overview and Motivation

This section looks at why and how to organise Fragments into *categories*. "Requirement", "domain knowledge", "specification", "goal", and so on, are examples of recurrent categories in RE. I focus on the following issues, moving from simpler to more complicated topics on categories.

1. Why and how to use independent categories? (Section 7.1)
2. What to do when there is a taxonomy of categories? (Section 7.2)
3. What is the meta-model, and what the ontology of a language? (Section 7.3)
4. Why and how to define derived categories and relations in a language? (Section 7.4)
5. How to enforce the intended use of categories in a language? (Section 7.5)

A category groups Fragments which share the same properties, and thus distinguishes these same Fragments from others which do not.

In absence of categories, it is not possible, for example, to make a language which represents instances of the Default RP. This is because the Default RP distinguishes three categories, namely, "requirement", "domain knowledge", and "specification". As I will argue below, categories cut up the information used in problem-solving, and thereby reflect the language designer's understanding of which way to cut up the information is useful to identify and solve RP instances. I will continue this argument much later, in Section 13, where I will discuss how the choice of categories enables a language to represent and solve some RP classes, and not others.

7.1 How to Have Independent Categories?

Categories are independent if, when adding them to a language, you do not *also* need to add new relations. This also means that, when there is a set of independent categories, you can choose any of its subsets to add to a language.

Categories in the Default RP are independent, even though they are used together in that problem, and even though that problem would not be the same if we removed any of these categories from it. They are independent, because whether a Fragment belongs to the "requirement" category is independent from there being the categories "domain knowledge" and "specification". This, in turn, is determined by how each of these categories is defined [140]:

> *"The primary distinction necessary for requirements engineering is captured by two grammatical moods. Statements in the 'indicative' mood describe the environment as it is in the absence of the machine or regardless of the actions of the machine; these statements are often called 'assumptions' or 'domain knowledge.' Statements in the 'optative' mood describe the environment as we would like it to be and as we hope it will be when the machine is connected to the environment. Optative statements are commonly called 'requirements.' [...] A specification is also an optative property, but one that must be implementable."*

Exercise 14: Define the minimal set of categories needed to represent those of the Default RP

> Define the minimal set of categories which a language would need, to make the distinctions suggested in the quote from Zave & Jackson. How many categories are needed? What are the properties which decide if a Fragment is in one of these categories? Can a Fragment be in two or more of these categories? If yes, which conditions does it have to satisfy? If not, then why not?

Below, I define each of the three categories as a Language Module. The Language Module has the same slots as for relations. This is because I see categories as being unary relations over the things that they categorise. However, because it should be clear when I am talking about relations, and when about categories, I call the modules "categories". Here is a definition of the requirement category, inspired by the definition of requirement in Default RP.

Category
Requirement (c.r)

Domain

c.Requirement ⊆ F, where F is a set of Fragments.

Membership conditions

x is in the optative mood, and describes "the environment as we would like it to be and as we hope it will be when the machine is connected to the environment" [140].

Reading

$x \in$ c.r reads "x is a requirement".

Language Services

- **s.IsReq**: Is x a requirement? Yes, if $x \in$ c.r.

The rules slot above carries over the informal definition from Zave & Jackson, that the requirement must be an optative statement. Following this same approach, there is a category for domain knowledge.

Category

Domain knowledge (c.k)

Domain

c.Domain knowledge ⊆ F, where F is a set of Fragments.

Membership conditions

x is in indicative mood and describes "the environment as it is in the absence of the machine or regardless of the actions of the machine" [140].

Reading

$x \in$ c.k reads "x is domain knowledge".

Language Services

- **s.IsDomK**: Is x domain knowledge? Yes, if $x \in$ c.k.

And finally, there is a category for specifications.

Category

Specification (c.s)

Domain

c.Specification ⊆ F, where F is a set of Fragments.

Membership conditions

x is a statement in optative, which is implementable, that is, it is known who and how will do what the statement says.

Reading

$x \in$ c.s reads "x is a specification".

Language Services

- **s.IsSpec**: Is x a specification? Yes, if $x \in$ c.s.

The three categories above can be used together, in a function that categorises sets of Fragments to deliver the following Language Service.

Language Service

WhichKSR: Which Fragments in X are requirements, which are domain knowledge, and which are specifications?

s.WhichKSR is similar to asking if one specific Fragment is in any of the three categories. Such questions are relevant when solving the Default RP, because we need to check, for example, if satisfying Fragments for domain knowledge and specifications, positively influences the satisfaction of requirements Fragments. The function below delivers s.WhichKSR.

Function

Categorise in Default RP categories
(f.cat.ksr)

Input

Set X of Fragments.

Do

For each $x \in X$:

- if x is in c.r, then let $cat(x)$ = c.r, else
- if x is in c.k, then let $cat(x)$ = c.k, else
- if x is in c.s, then let $cat(x)$ = c.s.

Output

Function *ksr*.

Language Services

- s.WhichKSR: Function *ksr* says, for each Fragment in X, if it is a requirement, domain knowledge, or specification.

Example 7.1. For illustration, below is the language L.Menkar, which has r.inf.pos, r.inf.neg, r.str.inf, and f.cat.ksr.

Language

Menkar

Language Modules

F, r.inf.pos, r.inf.neg, r.str.inf, f.map.abrel.g, f.cat.ksr

Domain

Fragments are partitioned onto requirements, domain knowledge, and specifications, that is, $\mathsf{F} = \mathsf{c.r} \cup \mathsf{c.k} \cup \mathsf{c.s}$ and $\mathsf{c.r} \cap \mathsf{c.k} \cap \mathsf{c.s} = \emptyset$. Influence relations are over Fragments, $\mathsf{r.inf.pos} \subseteq \mathsf{F} \times \mathsf{F}$, $\mathsf{r.inf.neg} \subseteq \mathsf{F} \times \mathsf{F}$. Relative strength of influence is a relation over influence relations of the same type:

$$\mathsf{r.str.inf} \subseteq (\mathsf{r.inf.pos} \times \mathsf{r.inf.pos}) \cup (\mathsf{r.inf.neg} \times \mathsf{r.inf.neg}).$$

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every

ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & r \mid k \mid s \\ \gamma & ::= & \beta(\alpha) \\ \delta & ::= & (\gamma, \gamma) \\ \epsilon & ::= & (\delta, \delta) \\ \phi & ::= & \gamma \mid \delta \mid \epsilon \end{array}$$

Mapping

α symbols denote uncategorised Fragments. γ symbols denote categorised Fragments, so $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, and $\mathscr{D}(s(\alpha)) \in$ c.s. δ symbols denote influence relations, $\mathscr{D}(\delta) \in$ r.inf.pos $\cup$ r.inf.neg. ϵ symbols denote relative strength of influence, $\mathscr{D}(\epsilon) \in$ r.str.inf.

Language Services

Those of r.inf.pos, r.inf.neg, r.str.inf, f.map.abrel.g, and f.cat.ksr.

Figure 9 is a visualisation of a model in L.Menkar. Label "R" marks c.r Fragments, "K" those of c.k, and "S" those of c.s. •

7.2 How to Define Taxonomies of Categories?

A taxonomy of categories is a set of categories related by the specialisation, also called is-a, relation. If a category A is a specialisation of the category B, then all members of B are also members of A, but not all members of A are necessarily members of B.

For illustration, note that it is common in RE to distinguish between functional and nonfunctional requirements. I can have two categories for them, both specialisations of c.r.

Exercise 15: Define categories which are specialisations of an existing category in a language

> Choose an existing category in L.Menkar and identify at least two categories which are its specialisations (its sub-categories). How would you define these new categories? How would you indicate in their definitions that they specialise another category of a language?

I will consider that a requirement is functional, if it can either be satisfied or not. A requirement is nonfunctional, if it can be satisfied to some extent, and different stakeholders may judge the requirement to be satisfied to different extents, by the same system. Being able to communicate via radio with an ambulance is a functional requirement, while quickly responding to incidents is a nonfunctional requirement.

Category

Functional requirement (c.r.f)

Domain

c.Functional requirement $\subseteq X$, where $X \subseteq$ c.r.

Membership conditions

x is a member of c.r such that it is either satisfied or not.

Reading

$x \in$ c.r.f reads "x is a functional requirement".

Language Services

- **s.IsFunctReq**: Is x a functional requirement? Yes, if $x \in$ c.r.f.

Category

Nonfunctional requirement (c.r.nf)

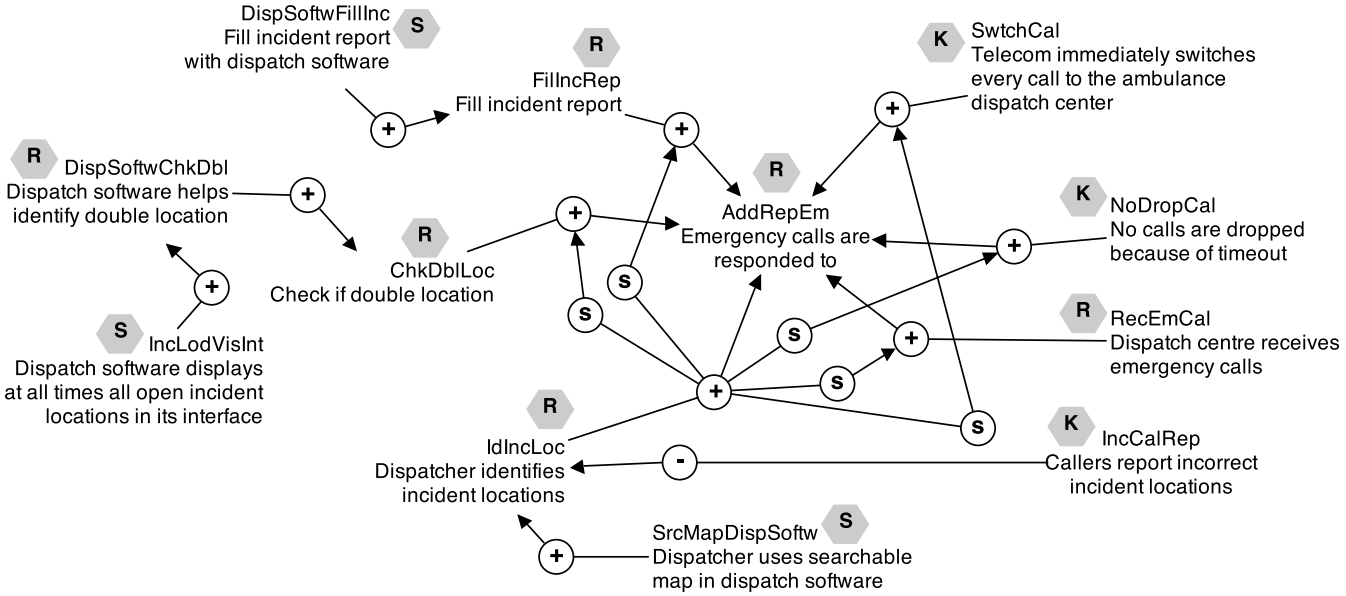


Figure 9: A visualisation of a model in L.Menkar.

Domain

c.Nonfunctional requirement $\subseteq X$, where $X \subseteq$ c.r.

Membership conditions

x is a member of c.r such that it is can be satisfied to some extent, rather than either satisfied or failed, and different stakeholders may judge it to be satisfied to different extents by the same system.

Reading

$x \in$ c.r.nf reads "x is a nonfunctional requirement".

Language Services

- **s.IsNFunctReq**: Is x a nonfunctional requirement? Yes, if $x \in$ c.r.nf.

Note that, if you let all Fragments be partitioned onto requirements, domain knowledge, and specifications, then the latter three categories are specialisations of a category for Fragments. You can see that Fragments category as the most general category, as shown in the taxonomy in Figure 10.

If a category is a specialisation of another one, then the former inherits the properties of the latter. Modules above captured inheritance by restricting domains, in that functional requirements are some of the requirements. This is clear from the slot "categorises" in the Language Modules above.

An important design decision concerns the coverage of the taxonomy. If c.r is specialised onto functional and nonfunctional requirements, are these its only subcategories? The taxonomy in Figure 10 says these are the only categories, but the Language Modules do not. To add this constraint, you could add a function to the language, which categorises any requirement either as a functional or a nonfunctional one.

7.3 What Are Categories and Relations in Meta-Models and Ontologies?

A *meta-model* is a conceptual model which represents all the categories and relations of a language. An *ontology* is a specification of a conceptualisation, and in RMLs, it is the specification of the categories and relations of the domain of the language, that the things in the domain that language expressions, the formulas, are used to represent. The categories and relations are chosen so as to help the representation and resolution of RPs [79]. In *formal ontology*, such a specification is written in a formal logic [58, 119, 120].

The meta-model and ontology of a language should not be confused [36]. The meta-model will usually represent also the considerations which are purely practical, and concern, for example, the structure of expressions in a language. In the terminology of the languages discussed in this tutorial, a meta-model will, for example, include a category "Graph", which may then be specialised into categories of graphs specific to each relation. However, the fact that graphs are used to represent relation instances is usually simply a practical matter, not something that fundamentally determines the conceptualisation of the requirements problems, which a language is defined for. In other words, a meta-model of the language may include all categories and relations from the ontology of the language, but will usually include also other categories and relations, concerned purely with practical issues of how to represent or do some transformations of the instances of the categories and relations in the ontology of the language.

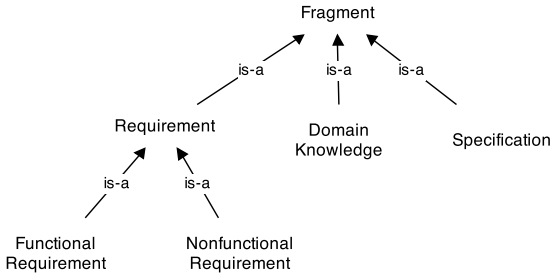
If a sufficiently expressive ontology specification language is used, it may be that the formal ontology of the language could define the language in its entirety. To the best of my knowledge, this has not been done in RE. The ontology of a language has usually been equated to the set of all categories and all relations in the language, together with axioms as constraints on how to correctly use the categories and relations. This is the case in *i**, *KAOS*, *Techne*, *NFR*, among others.

One way, then, to think of the ontology of an RML, is that it is the definition of the categories and relations needed to define instances of the requirements problem which the language is defined to solve, and potential solutions to these problems. For example, the definition of L.D1a is a specification of what that language is, and so, a specification of a conceptualisation. The other view, and more common in RE, is to see the ontology of the language only as all categories and relations of the language. In RMLs, this has often equated to a *meta-model* of the language, a conceptual model showing all categories and relations of the language. To represent the language ontology in such a way complements category and relation definitions with Language Modules, as Language Modules include information use rules and Language Services, which the said models do not represent.

Example 7.2. Figure 11(a) shows the categories and relations of three different languages, where for two, the Figure shows ontologies (Figures 11(a) and 11(b)), and for one, its meta-model (lower part of the Figure). Nodes represent categories and links represent relations.

Figure 11(a) shows the ontology of a language in which r.ifm, r.inf.pos, and r.inf.neg are over the members of any category in the taxonomy in Figure 10.

Figure 11(b) shows an ontology with same categories as in Figure 11(a), but now, the influence relations can go only from specification Fragments to requirement



[htb]

Figure 10: Taxonomy of categories defined in Sections 7.1 and 7.2.

Fragments.

Finally, Figure 11(c) shows a meta-model of a language, which has r.ifm, its specialisations, and the fragment, requirement, domain knowledge, specification, functional requirement, and nonfunctional requirement categories. Note the presence of "Relation", "Model" categories, which are interesting only for the creation of models in this language, but would normally not be part of the ontology of this language. •

7.4 When Are Categories and Relations Derived?

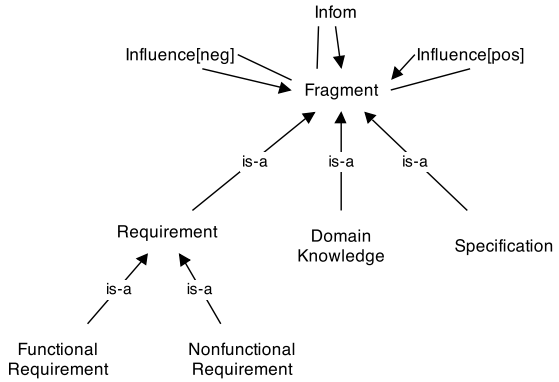
When new categories and relations are defined only as combinations of other parts of a language, I call them *derived*. Those which are not derived are called *core* language components. The core includes the minimal set of categories and relations, needed to define the others in that language. Derived relations will therefore inherit the properties of the core ones.

Derived categories and relations can be used to emphasise specific ideas in a language, or, for example, to simplify modelling. They are syntactic sugar in an RML. The following example illustrates this.

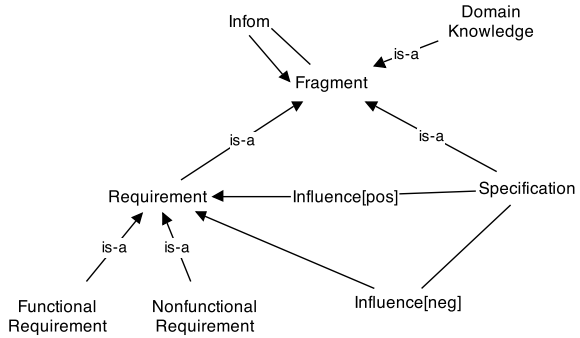
Example 7.3. Figure 11 shows that there can be an influence relation over problem-solving information, and consequently, over any pair of Fragments, regardless of either of them being a requirement, domain knowledge, or otherwise. If you want to emphasise that there is a difference between having an influence relation from a specification to a requirement, as opposed to having it between requirements, you can add a derived relation as follows. Call it r.rls.

Relation
Realise (r.rls)
Domain & Dimension r.rls $\subseteq S \times R$, where $S \times R \subseteq$ r.inf, $S \subseteq$ c.s and $R \subseteq$ c.r.
Properties irreflexive and transitive.
Reading $(x, y) \in$ r.rls reads "specification x realises the requirement y".
Language Services Inherits from r.inf.

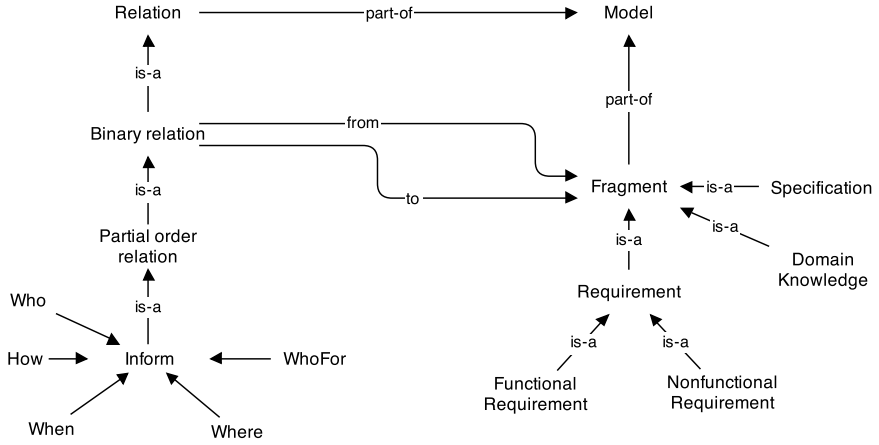
r.Realise is the abbreviation of an influence from a specification to a requirement. You may want to distinguish r.rls from others in an RML, because there may be guidelines which rely on it, and so it may be simpler to talk of realisation every time the guidelines are applied, rather than of all that it abbreviates. Or, it may be that



(a) Ontology of one language.



(b) Ontology of another language.



(c) A meta-model of a language.

Figure 11: Two ontologies and a meta-model.

there is a convention among stakeholders, who speak of requirements being realised or not, and you interpret this as being about the presence or absence of influence relations from specifications to these requirements. •

A derived category can be defined from categories and relations only, but also from combinations of other language components, categories and functions for example. I look at the former first.

Example 7.4. Suppose that I am particularly interested in requirements which are negatively influenced by environment conditions. If I assume that I cannot change the environment conditions, then such requirements will likely need to be revised, to avoid that the system fails them too often at run-time. To highlight them in models, I define a new derived category c.r.clsh. •

Category
Clashing requirement (c.r.clsh)
Domain c.Clashing requirement $\subseteq X$, where $X \subseteq$ c.r.
Membership conditions x is such that there is $(y, x) \in$ r.inf.neg, and $y \in$ c.k.
Reading $x \in$ c.r.clsh reads "x is a requirement which clashes with environment conditions".
Language Services • **s.IsClshReq**: Does x clash with environment conditions? Yes, if $x \in$ c.r.clsh.

Example 7.5. Now suppose that I want to categorise a requirement as irrelevant, if that requirement is not acceptable. Acceptability works as in Section 5.4.2. I use f.Accepted to define the category c.r.irrl. •

Category
Irrelevant requirement (c.r.irrl)
Domain c.Irrelevant requirement $\subseteq X$, where $X \subseteq$ c.r.
Membership conditions x is not acceptable in a given model M according to f.acc.
Reading $x \in$ c.r.irrl reads "x is an irrelevant requirement".
Language Services • **s.IsIrrlReq**: Is x an irrelevant requirement? Yes, if $x \in$ c.r.irrl.

7.5 How to Enforce Intended Use of Categories?

Categories are interesting because they distinguish Fragments in terms of how they are used in problem-solving. So categorising a Fragment is only part of how categories are used. The other part is to define rules about how to use these categories. This can, for example, be functions which say what to do, when there is an instance of some category, or if instances of a category are in some specific relations with instances of other categories.

Another way to view this, is that you are adding new functions to a language, in order to make sure that the categories in it are used as you intended. In the following example, I use c.r as a completeness check of models.

Example 7.6. Knowing that a Fragment is a requirement leads me to ask if this requirement is operationalised in the given model model. If it is not, then I might

want to conclude that this is negative, and say that the model is incomplete. If I want to force this notion of model completeness on language users, I can build it into the language with the following function.

Function

Completeness of requirements operationalisation
(f.chk.rop)

Input

A set X of Fragments, G(X, r.ifm), and G(X, r.inf.pos).

Do

1. Let H be a hypergraph made by merging G(X, r.ifm) and G(X, r.inf.pos).
2. If there is $x \in X$ such that x is in c.r and there is no path in H from $z \in X$ to x, such that z is in c.s, then the model which includes exactly the Fragments in X is incomplete with regards to requirements operationalisation and $v = 1$.

Output

v.

Language Services

- **s.IsROpComp**: Is the model that includes exactly the Fragments X incomplete with regards to requirements operationalisation? : Yes, if $v = 1$, no otherwise.

I can use f.chk.rop as a way to check how close we are to identifying a solution to the RP being solved. If some requirements are not operationalised, then I have to look further for specifications, as I have not solved the problem yet. •

7.6 Summary on Categories

The following are the main ideas discussed on categories:

- To add some category C to a language, it is necessary to define how it is used. At the very least, this involves answering the following questions:
  1. What conditions have to be satisfied for x to belong to the category C?
  2. Can members of C be members of other categories in the given language? If yes, then why and of which categories? This is answered by defining taxonomic (is-a) relations between categories.
  3. How are category instances, if in any way, related to those of other categories?
- Using categories for classification is only part of the motivation for having them in languages. After adding a category, such as c.r, you may want to add new relations, functions, and so on, in order to use that category in problem-solving. For example, having a category for requirements and for specifications begs the question of how the satisfaction of the latter influences that of the former, and to answer it, you need influence relations. Having domain knowledge and requirements categories begs the question of what to do if there is negative influence from the latter to the former, and so requires guidelines for resolving this.
- It is useful to distinguish core categories and relations from derived ones in a language. It is otherwise hard to know what is absolutely necessary in a language, in order to deliver Language Services, as well as to compare languages in terms of their components.

8 Alternatives and Combinations

Overview and Motivation

This section is on how to represent *mutually exclusive* information in models. I look at two specific notions. One, called "Alternative", allows me to represent that, say, two relation instances are mutually exclusive. The other, called "Combination", lets me say that sets of Fragments and, or relation instances, are mutually exclusive. I discuss the following questions.

1. How to represent simple Alternatives, where an Alternative equates to a smallest part of a model? (Section 8.1)
2. How to represent composite Alternatives in models, that is Combinations, where a Combination can include various model parts? (Section 8.2)
3. How to find all Combinations in a model which includes Alternatives? (Section 8.3)

If you want to represent different ways to satisfy a requirement, or to solve a conflict between requirements, or entire designs which, for example, you want stakeholders to choose from, then it is necessary to have a language that can represent mutually exclusive information.

Exercise 16: Represent that two relation instances are mutually exclusive

Choose any language defined so far in this tutorial. Without changing that language, how would you represent that two relation instances are mutually exclusive in a model in that language?

For example, there are different ways to fill out an incident report. It can be printed on paper and manually filled out, or there could be a template document of the report for use in word processing software, or by having a dedicated functionality for this in the dispatch software, or in some other way. For each of these, you can probably think of alternative organisational positions whom this responsibility can be assigned, such as dispatcher or administrative assistant, for example.

To represent different ways of doing FillIncRep, and do so with languages defined so far, I would have to make one model per Alternative. This is impractical, because suppose that there are three different ways to fill out a report, and two ways to allocate responsibility for doing so. This gives gives eight alternatives, and they cover only some options and only for FillIncRep, not other Fragments. Moreover, if the RML cannot represent alternatives, it will not be able to represent relations between alternatives. So there can be many models, one per alternative, but no information in the same language, about which alternative is, for example, more desirable than another one over some criterion, such as cost to implement.

Problem-solving involves making decisions, that is, given various possible ways to act, committing to one only. A basic notion of decision-making is that of an *Alternative*. Some x, whatever it may be, can be called an Alternative when there are $m \geq 1$ other things, say $y_1, \ldots, y_m$ that can perform the role of x, we have the ability to choose any of $x, y_1, \ldots, y_m$ for that role, and $x, y_1, \ldots, y_m$ are mutually exclusive, that is, neither is compatible with others, and neither is part of another.

To use models for decision-making, it is necessary to be able to represent Alternatives and to represent relations between them. The model becomes a record of Alternatives which were encountered during problem-solving. This allows you to postpone choosing any one Alternative before discovering others and comparing them. For example, I may want to postpone choosing an Alternative, because I need more information to find others, or I need to present the Alternatives to stakeholders who have the authority to decide, or I want first to find criteria for the comparison of the alternatives (more on this in Section 11), before doing anything else with them.

8.1 How to Represent and Use Simple Alternatives?

In languages discussed so far, individual Fragments and individual relation instances were the smallest useful parts of models. I focus in this section on how to represent that these smallest model parts are Alternatives. To illustrate this, suppose that you want a language to deliver the following Language Service.

Language Service

InfPosAlt: Which Fragments in the model M show different ways to satisfy x?

L.Ankaa can show positive and negative influence relations over Fragments. It can show that different Fragments positively influence some Fragment x. But it cannot show that these Fragments are mutually exclusive ways to satisfy x.

Exercise 17: Make a new language from L.Ankaa which can show mutually exclusive relation instances in models

How would you represent in models that two or more relation instances are mutually exclusive? Would you do it with a relation, or otherwise? What would you add or remove from L.Ankaa to enable it to show in models that some influence relation instances are mutually exclusive? Would the resulting language deliver s.InfPosAlt? If yes, then how?

To deliver s.InfPosAlt, notice first that it is not the Fragments $x_1,\dots,x_n$ themselves which are mutually exclusive, but the instances of the positive influence relation. s.InfPosAlt is not about choosing one Fragment, but about choose which of these Fragments should positively influence x.

A relation instance becomes an Alternative by being related in some way to other relation instances, all being mutually exclusive. This relation will be called r.xor. When it is over n other relation instances, it says that these are mutually exclusive. I define it as follows.

Relation

Mutually exclusive relation instances (r.xor)

Domain & Dimension

r.xor $\subseteq R^n$, where R is a set of relation instances.

Properties

If $w \in$ r.xor, then there can be no $e \in$ r.xor such that $e = (\dots, w, \dots)$, that is, there can be no r.xor instances over other r.xor instances.

Reading

$(r_1,\dots,r_n)$r.xor reads "relation instances $r_1,\dots,r_n$ are mutually exclusive".

Language Services

- **s.IsAlt**: Are $r_i,\dots,r_m$ Alternatives? : Yes, if there is $w \in$ r.xor and every $r_i,\dots,r_m$ is in w.

Example 8.1. I define below the language L.Mirfak, which can represent positive and negative influence over requirements, domain knowledge, and specifications, just as L.Menkar. It differs from L.Menkar in that it does not represent strength of influence, but can also represent mutually exclusive influence relation instances. The omission of strength of influence is motivated by simplicity in visualisations, in Figures.

Language

Mirfak

Language Modules

F, r.inf.pos, r.inf.neg, r.xor, f.cat.ksr, f.map.abrel.g

Domain

Fragments are partitioned onto requirements, domain knowledge, and specifications, F = c.r $\cup$ c.k $\cup$ c.s and c.r $\cap$ c.k $\cap$ c.s = $\emptyset$. Positive and negative influences are over Fragments, r.inf.pos $\subseteq$ F $\times$ F, r.inf.neg $\subseteq$ F $\times$ F. r.xor is over $n > 1$ influence relation instances of the same type,

$$\text{r.xor} \subseteq (\text{r.inf.pos}^n) \cup (\text{r.inf.neg}^n).$$

Syntax

A model M in the language is a set of symbols $M = \{\phi_1,\dots,\phi_n\}$, where every

ϕ is generated according to the following BNF rules:

$$
\begin{array}{lcl}
\alpha & ::= & x \mid y \mid z \mid \ldots \\
\beta & ::= & r \mid k \mid s \\
\gamma & ::= & \beta(\alpha) \\
\delta & ::= & (\gamma, \gamma) \\
\epsilon & ::= & (\delta, \delta, \ldots) \\
\phi & ::= & \gamma \mid \delta \mid \epsilon
\end{array}
$$

Mapping

α symbols denote Fragments, and $r(\alpha)$ denote requirements $\mathscr{D}(r(\alpha)) \in$ c.r, $k(\alpha)$ denote domain knowledge $\mathscr{D}(k(\alpha)) \in$ c.k, and $s(\alpha)$ denote specifications, $\mathscr{D}(s(\alpha)) \in$ c.s. δ denote positive and negative influence relation instances, $\mathscr{D}(\delta) \in$ r.inf.pos $\cup$ r.inf.neg. ϵ symbols denote r.xor instances, $\mathscr{D}(\epsilon) \in$ r.xor.

Language Services

s.IsAlt.

Figure 12 shows a visualisation of a model made with L.Mirfak. There, if the circle is labelled "+", then it is an instance of r.inf.pos, and if "-" then of r.inf.neg. A black circle labeled "X" and its dashed lines represent an instance of r.xor. All relation instances connected to a black circle via dashed lines are Alternatives.

The content of the model comes from the CADS example, and looks at how to satisfy the requirement AmbArrIncLoc. The model has 14 instances of r.inf.pos, no instances of r.inf.neg, and six instances of the meta-relation r.xor.

Consider first the following two r.xor instances:

((AutoAmbList, IdAmb), (ManTrckAmb, IdAmb)) ∈ r.xor

((UpdAutoAmbList, IdAmb), (ManTrckAmb, IdAmb)) ∈ r.xor

Together, this pair of r.xor shows two Combinations to satisfy (to influence positively, but I will say satisfy for simplicity) IdAmb. One option consists of doing according to AutoAmbList together with UpdAutoAmbList. The other is to do according to only ManTrckAmb.

The remaining four r.xor instances are:

((DispSoftwRnkAmb, ChoAmb), (NoAutAmbRnk, ChoAmb) ∈ r.xor

((DispSoftwRnkAmb, ChoAmb), (NoAmbRecomm, ChoAmb) ∈ r.xor

((DispAmbRnk, ChoAmb), (NoAutAmbRnk, ChoAmb) ∈ r.xor

((DispAmbRnk, ChoAmb), (NoAmbRecomm, ChoAmb) ∈ r.xor

Given these four instances, there are two Combinations for satisfying ChoAmb. One includes DispSoftwRnkAmb, DispAmbRnk, and CAsstChoAmb. The other includes CAsstChoAmb, NoAutAmbRnk, and NoAmbRecomm.

In total, then, the model visualised in Figure 12 shows four Combinations. Section 8.3 has more on Combinations. •

8.2 How to Have Alternative Composites?

How to represent that sets of relation instances are mutualy exclusive? For example, in Figure 12, I could not say that the *joint* satisfaction of CIlncRep, ConfMob, MobAmb, AsAmb, ChoAmb, and IdAmb positively influences the satisfaction of AmbArrIncLoc. All that I could say with positive influence relations, is that the satisfaction of each of these, *independently of others*, positively influences the satisfaction of AmbArrIncLoc.

Also in Figure 12, I had four r.xor instances over the positive influence relations targeting ChoAmb, to convey, only in part, the idea that there are two ways to satisfy ChoAmb. One is described by DispSoftwRnkAmb, DispAmbRnk, and CAsstChoAmb together, and the other by CAsstChoAmb, NoAutAmbRnk, and NoAmbRecomm together.

If I could say in a model that Fragments belongs to some set, then I can define relations over such sets. There could, for example, be a positive influence relation from a set of Fragments to a single Fragment, which would help in the first case above. I could also define a relation that makes sets into Alternatives, and I would need only one such relation instance in the second case, over the set that includes DispSoftwRnkAmb, DispAmbRnk, and CAsstChoAmb, and the set with CAsstChoAmb, NoAutAmbRnk, and NoAmbRecomm.

Exercise 18: Compose Fragments and relation instances into wholes

Choose any language defined so far. What do you need to add to that language, in order to represent that some Fragments and, or relation instances are parts of some whole? Would you do it with one or more new relations? If yes, which? Do you need one or more new categories? If yes, which?

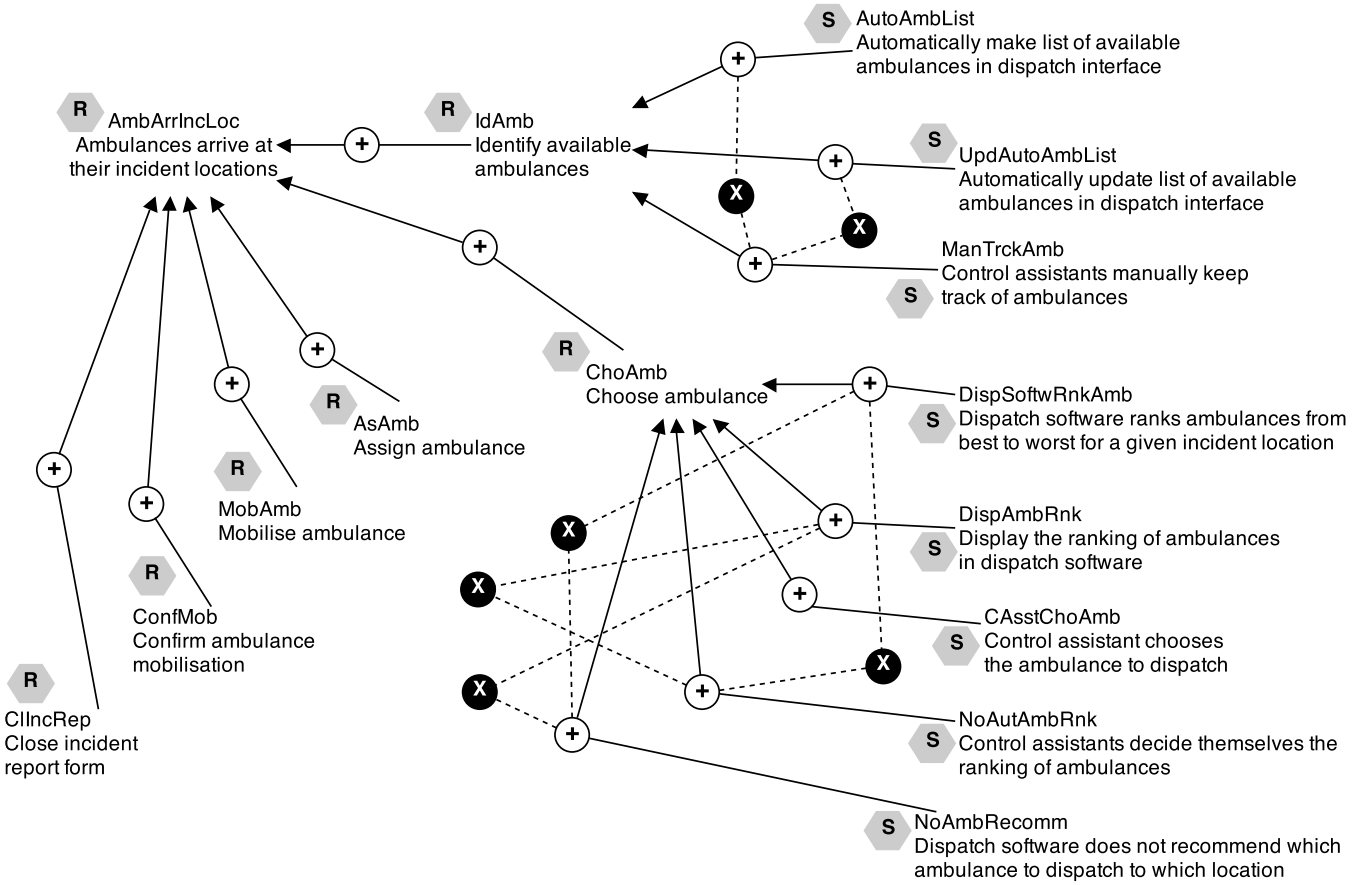


Figure 12: A visualisation of a model in L.Mirfak.

I will use the term *Composite* and say that Fragments can be *parts of* a Composite. The smallest part of a Composite is a single Fragment, and a Fragment can be in more than one Composite. This lets me represent, for example, Alternatives which overlap. An example is CAsstChoAmb in Figure 12, which is there both with DispSoftwRnkAmb and DispAmbRnk, and with NoAutAmbRnk and NoAmbRecomm. As Composites can overlap, and individual Fragments can be Composites themselves, it follows that Composites can be parts of other Composites.

Enabling the above requires a relation between a Composite and its parts, which will be called r.po, and a category for Composites, c.Composite.

r.po is a partial order. It is reflexive, because a Composite can include a single Fragment. It is antisymmetric, because Composites with same parts should be considered as same Composites. It is transitive, because there is no reason why we should not conclude that if a Fragment x is part of Composite y, and y is part of Composite z, that x is not part of z. With these formal properties, r.po fits mainstream theories in mereology [133]. This gives the following r.po definition.

Relation

Fragment part of Composite (r.po)

Domain & Dimension

r.po $\subseteq$ F $\times$ C, where F is a set of Fragments and C is a set of Composites.

Properties

reflexive, antisymmetric, and transitive.

Reading

$(x, c) \in$ r.po reads "x is part of c".

Language Services

- **s.IsPartOf**: Is x part of c? : Yes, if there is $(x, c) \in$ r.po.
- **s.HasParts**: Does x have parts? : Yes, if there is $(y, x) \in$ r.po.

A Composite will be any Fragment which has parts. x "has parts" when there is at least some other y such that there is also $(y, x) \in$ r.po.

Category

Composite (c.cp)

Domain

c.Composite $\subseteq F$, where F is a set of Fragments.

Membership conditions

There is $y \in F$ such that $(y, x) \in$ r.po.

Reading

$x \in$ c.cp reads "x is a Composite".

Language Services

s.HasParts.

The category c.cp is a specialisation of c.Fragment, as it is a Fragment with parts. The example below illustrates these Language Modules in a language.

Example 8.2. The following language adds r.po and c.cp to L.Mirfak.

Language

Aldebaran

Language Modules

F, r.inf.pos, r.inf.neg, r.xor, f.cat.ksr, f.map.abrel.g, r.po

Domain

Same as L.Mirfak, and add r.po $\subseteq$ F $\times$ C and $C \subset$ F.

Syntax

Same as L.Mirfak.

Mapping

Same as L.Mirfak, and replace $\mathscr{D}(\delta)$ with

$$\mathscr{D}(\delta) \in \text{r.inf.pos} \cup \text{r.inf.neg} \cup \text{r.po}.$$

Language Services

s.IsAlt, s.IsPartOf, s.HasParts.

Figure 13 shows a visualisation of a model in L.Aldebaran. If the circle is labeled "Po", it represents an instance of r.po. c1, c2, and c3 are the only Composites. Other symbols read as in Figures used earlier in this tutorial.

Compare the model in Figure 13 to that in Figure 12. The L.Aldebaran model shows the same Combinations, but now includes three new Fragments, c1, c2, and c3. Each of them is a Composite. The main difference between the two models, is that Figure 13 says that Alternatives are combinations of influence relation instances, rather than individual relation instances, and thereby lets me convey a bit more closely the idea that, for example, the satisfaction of ChoAmb is positively influenced by the satisfaction of DispSoftwRnkAmb, DispAmbRnk, and CAsstChoAmb together. •

L.Aldebaran illustrates that having Composites, r.po, and r.xor allows the language to represent Alternative Composites.

8.3 What Are and How to Find Combinations?

Given some sets $A_1, \ldots, A_n$, such that each A_i is a set of Alternatives, that is, of mutually exclusive things, a *Combination* is a set C obtained as follows: in each $A_1, \ldots, A_n$, pick *exactly one* member and add it to C.

Exercise 19: Define a procedure which finds all Combinations in a L.Mirfak

Define a new category in L.Mirfak for Combinations. Define a procedure which takes a model in L.Mirfak and returns the set of all Combinations in that model.

Each member of a Combination is also called a Choice. It is called a Choice, because it is one Alternative, which was picked out from a set of Alternatives.

Example 8.3. In Figure 12, there are six instances of r.xor. Each instance gives a set of Alternatives, and each of these sets includes two Alternatives. In the terminology above, the figure shows the following:

$$\begin{aligned}
A_1 &= \{ (\text{AutoAmbList}, \text{IdAmb}), (\text{ManTrckAmb}, \text{IdAmb}) \} \\
A_2 &= \{ (\text{UpdAutoAmbList}, \text{IdAmb}), (\text{ManTrckAmb}, \text{IdAmb}) \} \\
A_3 &= \{ (\text{DispSoftwRnkAmb}, \text{ChoAmb}), (\text{NoAutAmbRnk}, \text{ChoAmb}) \} \\
A_4 &= \{ (\text{DispSoftwRnkAmb}, \text{ChoAmb}), (\text{NoAmbRecomm}, \text{ChoAmb} \} \\
A_5 &= \{ (\text{DispAmbRnk}, \text{ChoAmb}), (\text{NoAutAmbRnk}, \text{ChoAmb}) \} \\
A_6 &= \{ (\text{DispAmbRnk}, \text{ChoAmb}), (\text{NoAmbRecomm}, \text{ChoAmb}) \}
\end{aligned}$$

A_1 includes two Alternatives. If I choose (AutoAmbList, IdAmb) in A_1, then that is my Choice with regards to A_1. Some Combination C_i is, for example:

$$\begin{aligned}
C_i = \{&(\text{AutoAmbList}, \text{IdAmb}), (\text{UpdAutoAmbList}, \text{IdAmb}), (\text{DispSoftwRnkAmb}, \text{ChoAmb}), \\
&(\text{DispAmbRnk}, \text{ChoAmb})\}
\end{aligned}$$

C_i was made by taking one Alternative from each of $A_1, \ldots, A_6$, that is, by choosing one Alternative in each of these sets. •

I define Combination as a new category, as follows. Example 8.4 further illustrates Combinations.

Category

Combination (c.cb)

Domain

c.Combination $\subseteq R$, where $\wp(R)$ is a set of sets of relation instances.

Membership conditions

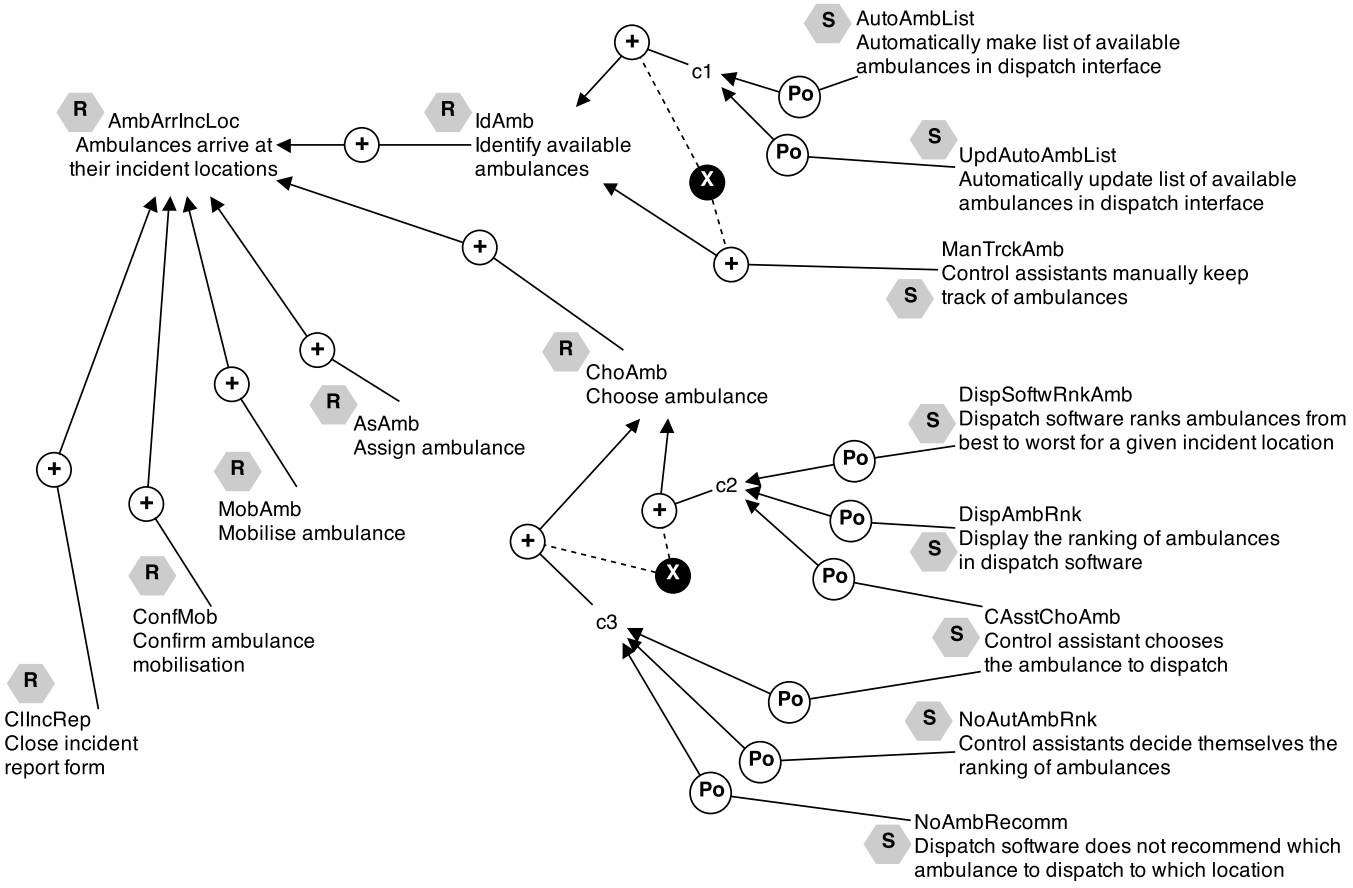


Figure 13: A visualisation of a model in L.Aldebaran.

If $W \in$ c.cb , then there is for every $w \in W$ an instance $r \in$ r.xor, such that w is one of the Alternatives in r, and no member of $W \setminus \{w\}$ is also in r.

Reading

$W \in$ c.cb reads "W is a Combination".

Language Services

- **s.IsCombination**: Is x a Combination? : Yes, if $x \in$ c.cb.

Example 8.4. Figure 14 shows all four Combinations in the model in Figure 12. The bold black relation instances are those that the Combination does *not* include. This shows what is removed from the model Figure 12, in order to obtain the respective Combination. When the bold relation instances are excluded, each Combination satisfies the rules in c.cb. •

Having clarified the notions of Alternative, Choice, and Combination, the aim now is to enable the following Language Service.

Language Service

AllCombinations: Which are all the Combinations in model M?

Let a model M be a triple (X, R, A), where X is a set of Fragments, R a set of relation instances which cannot include r.xor instances, and A is a set of r.xor instances, each over the members of R. For such models, I can deliver s.AllCombinations by generating sub-models of M, each of which is exactly one Combination, that is, a sub-model in which there are no r.xor instances (its set A is empty). f.find.all.cb does this.

Function

Find all Combinations
(f.find.all.cb)

Input

$M = (X, R, A)$, where X is a set of Fragments, R a set of relation instances which cannot include r.xor instances, and A is a set of r.xor instances, each over the members of R.

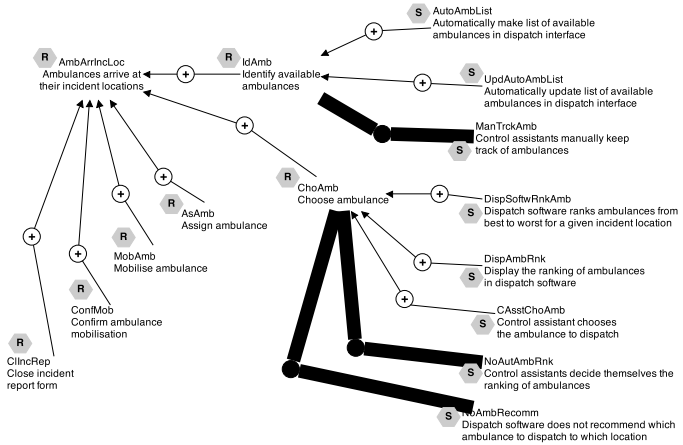
Do

Let

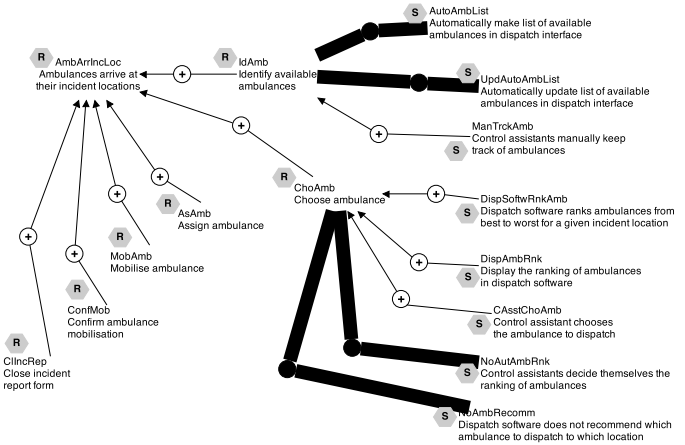
- a be some (any) member of A,
- $Alt(a) \subseteq R$ be the set of relation instances from R which are all relata of a (that is, a is over all members of $Alt(a)$). For example, if $a = (r_1, \ldots, r_m)$, and $\{r_1, \ldots, r_m\} \subseteq R$, then $Alt(a) = \{r_1, \ldots, r_m\}$,
- $a(r_i)$ be some (any) member of $Alt(a)$, that is, one of the Alternatives according to a,
- O be an empty set, in which every member will be a Combination from M,
- $M = (X_M, R_M, A_M)$ be the only member of a set Q. I write X_M, R_M, A_M, to make it clear that they are those of M.

For each member G of Q:

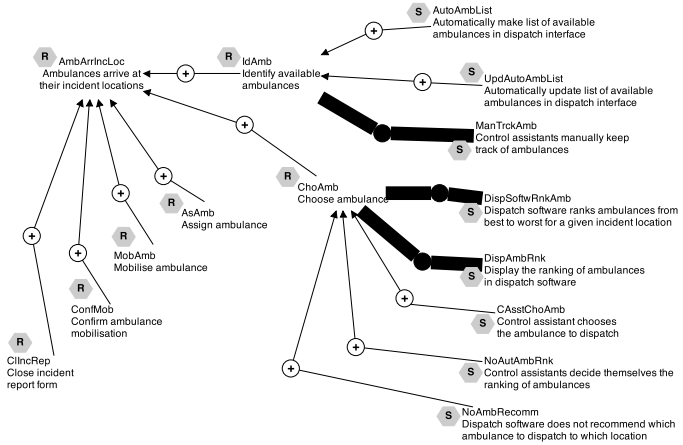
- if X_G is empty, then G includes no r.xor instances and is a Combination, so remove G from Q and add G to O,
- else, for each r.xor instance $a \in A_G$:
  - For each $a(r_i) \in Alt(a)$:
    1. Let $G(a(r_i)) = (X, R_G(a(r_i)), A_G(a))$ where:
       - $R_G(a(r_i)) = R_G \setminus (Alt(a) \setminus \{A(r_i)\})$, so $R_G(a(r_i))$ has all relations that R_G has, minus all but one relation, r_i, which is an Alternative according to a, and
       - $A_G(a) = A_G \setminus \{a\}$, so $A_G(a)$ includes all r.xor instances that A_G does, except for a;
    2. If $G(a(r_i))$ includes one or more r.xor instances, each of which has $a(r_i)$ as one of its Alternatives, then remove also all of these r.xor instances from $G(a(r_i))$ and remove all Alternatives except $a(r_i)$ in all these r.xor instances;



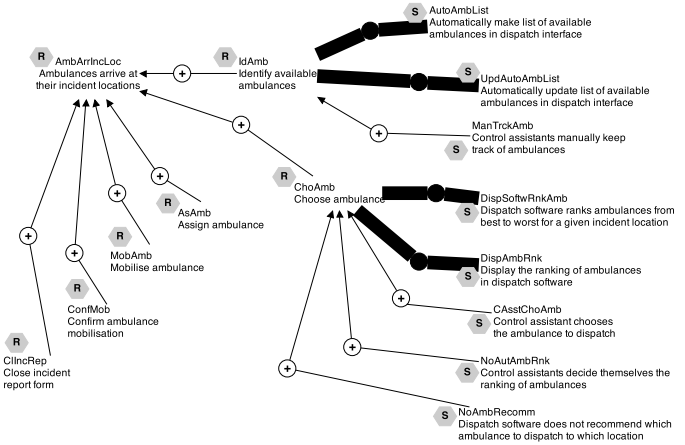
(a) One of four Combinations in Figure 12.



(b) A second of four Combinations in Figure 12.



(c) A third of four Combinations in Figure 12.



(d) A fourth of four Combinations in Figure 12.

Figure 14: Illustration of Combinations in a model.

3. Add $G(a(r_i))$ to Q.

Output

The set O, which includes exactly all the Combinations in M.

Language Services

- s.AllCombinations: Set O which this module outputs.

In f.find.all.cb, I start from the model M whose Combinations I want to identify. I take M, choose one r.xor instance in it, say a, and choose one of the Alternatives in it, say r_i. I delete all other relation instances in a, to convey the idea that the resulting $G(a(r_i))$ is the model I get, when I have made a decision about which Alternative to adopt among the Alternatives according to a. Because I add $M(a(r_i)$ to Q, I will process it in the same way, until I get a Combination, which is a model without r.xor instances. The intuitive idea is that I take a model, then generate its sub-models, by making decisions about each of its r.xor instances, and I produce sub-models in such a way that no Combination is missed. The example illustrates this.

Example 8.5. Figure 15 shows the result of applying f.find.all.cb to the model in Figure 12. In Figure 15, the rectangles labeled Q are models in the set Q, and those labeled O are in the set of Combinations. In each model in the Figure, all bold and black relation instances are *not* (are excluded from) that model (it is as if we marked the relation instances which we removed from the model).

The figure shows that I start by adding that model to the set Q, and then choose one of the r.xor instances.

In the Figure, I assume that the instance I chose is the one over (AutoAmbList, IdAmb) and (ManTrckAmb, IdAmb). Since that r.xor instance is over two Alternatives, I have two new models to add to Q. In one, shown in the second row of the first column in Figure 15, I removed (ManTrckAmb, IdAmb). Since (ManTrckAmb, IdAmb) is an Alternative also in another r.xor instance, over (ManTrckAmb, IdAmb) and (UpdAutoAmbList, IdAmb), removing (ManTrckAmb, IdAmb) leads me to remove that second r.xor instance as well.

The first row and second column in the Figure shows the case where I removed (AutoAmbList, IdAmb) from the r.xor instance over (AutoAmbList, IdAmb) and (ManTrckAmb, IdAmb). There, by leaving (ManTrckAmb, IdAmb), I also made the decision to remove (UpdAutoAmbList, IdAmb), because the r.xor instance over (ManTrckAmb, IdAmb) and (UpdAutoAmbList, IdAmb), has only these two Alternatives.

This same approach applies in the other cases shown in the Figure, until I find a Combination, and put it in the set O.

Be careful to note that the starting model, from Figure 12 has r.xor instances each time over only two Alternatives, and since some Alternatives participate in more than one r.xor, I quickly find the Combinations, that is, via few members of Q.

The two cases marked "Error" are there to show what would have happened, if f.find.all.cb did not remove r.xor instances, in cases when choosing one Alternative in some r.xor r_1 has the consequence that there is only one Alternative left in another r.xor instance r_2. That is, these are errors which I would have, if f.find.all.cb did not have the line 2 in its use rules. •

8.4 Summary on Alternatives

A language that aims to support design may need to represent alternative design options in models, via Alternatives, Choices, and Combinations. This section illustrated that discovery and indecision in problem-solving make this a relevant capability for a language.

Enabling a language to represent Alternatives raises many challenges, and this section focused on the basic ones. Namely, how to represent simple Alternatives, each being a single smallest part of a model, and how to represent complicated Alternatives, which are over composites, each made out of potentially many smallest parts of a model. Other topics which I did not discuss include:

- How is the presence of r.xor in a language related to the assignment of satisfaction, or other kinds of values to model parts? I discussed Alternatives and Combinations by talking about satisfaction, and it made sense to say, for example, that Alternatives are mutually exclusive because they cannot be satisfied together. But what happens if there are other values to assign? Do r.xor instances somehow influence the assignment of values to model parts? Does it matter that two model parts are mutually exclusive when assigning values to them? I discuss these questions in Section 9.

- How to compare Alternatives and Combinations, in order to choose "the best one"? I come back to this in Section 11.

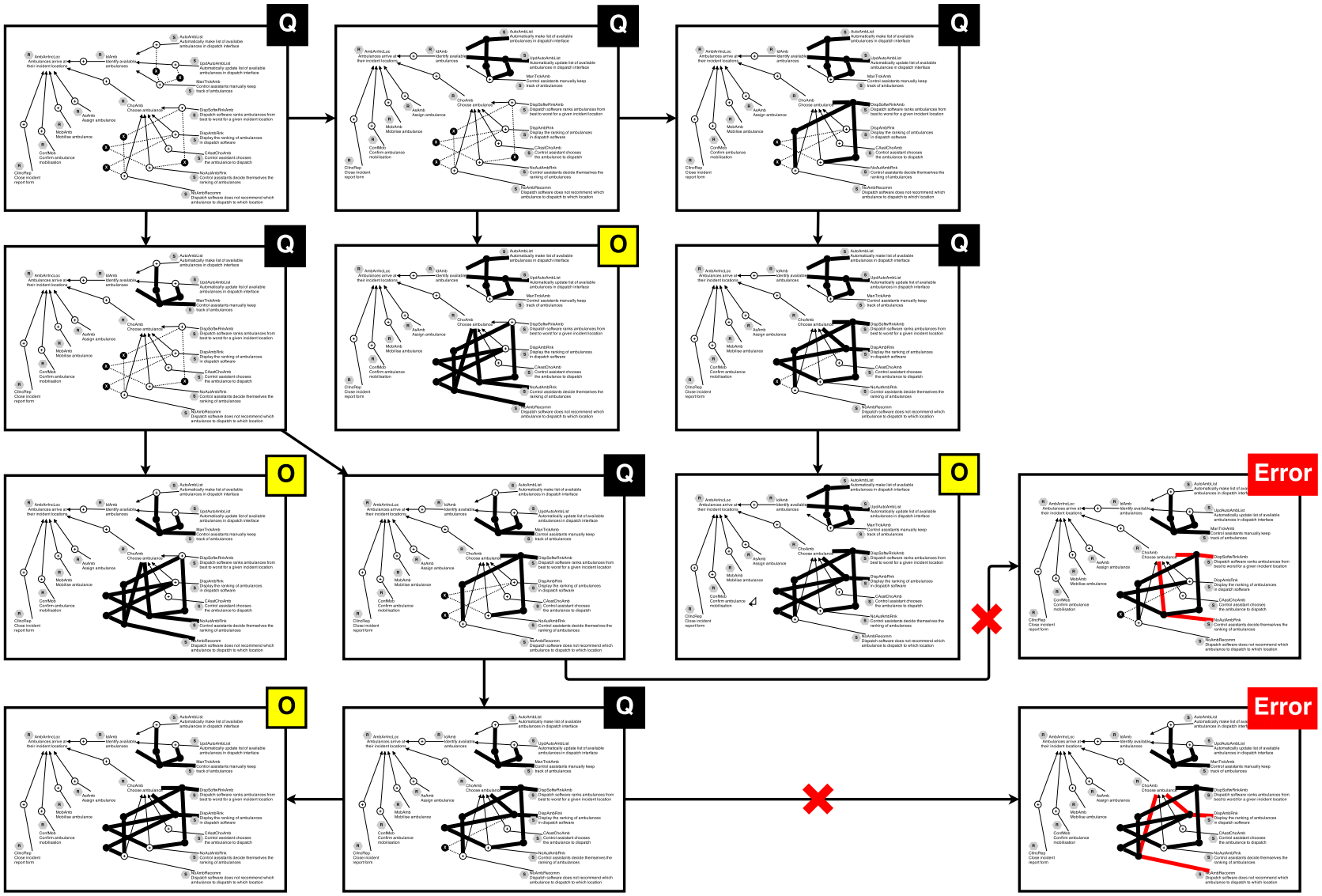


Figure 15: Application of f.find.all.cb to a model from Figure 12.

9 Valuation

Overview and Motivation

Valuation consists of associating variables to model parts, and functions to relations over the model parts. The aim is to have models, where values of some variables depend on values of others. Given the values of some, you can then compute those of others.

Value Type is a central notion in valuation. A Value Type is simply a set of values. I will write below that *a variable x has Value Type T*, if and only if any value of that variable must be a member of T. That is, when a value is assigned to x, that value is one of those in T.

The section looks at different Value Types and their combined use in a language. This is done by discussing the following questions.

1. How to define a language with a single binary Value Type, that is, where model parts take either of two values, and why this may be interesting? (Section 9.1),
2. How to, and why define a language which has more than one binary Value Type, so that any model part is assigned a tuple of values, instead of a single value? (Section 9.2),
3. How to, and why define a language with an unordered set of values as its only Value Type? (Section 9.3),
4. What if the value type for the language is an ordered set? (Section 9.4),
5. Why and how to have in a language, a Value Type defined over real numbers? (Section 9.5).

These discussions will also illustrate how to use Value Types in new guidelines for a language.

Valuation can enable various interesting Language Services. It is also related to Language Services discussed earlier. For example, valuation in a language can say that each Fragments is associated with its own variable for satisfaction. The variable might be allowed to take either 1, read "satisfied", or 0 for "not satisfied". A function may then be associated to every positive and negative influence relation, to compute how the value of the influenced Fragment depends on those of Fragments influencing it. This section will give many examples unrelated to satisfaction, but satisfaction remains an important motive to think about valuation in a language.

In this tutorial, a language has rules for valuation if, in its models, variables can be associated to model parts, functions to relation instances, and if that language answers the following questions:

1. Which values can be assigned to which variables?
2. Which functions relate the values of the variables?
3. How to compute the values of the variables?

This section will illustrate how to answer the questions above for various Value Types.

9.1 How to Propagate Binary Satisfaction Values in a Model?

This section discusses and combines three topics:

- How to have a language in which any Fragment and relation instance can be assigned one of two satisfaction values, namely satisfied or not satisfied? That is, how to define a language which has only one binary Value Type? (Section 9.1.1)
- Given a model in that language, and knowing the satisfaction value of some Fragments and, or relation instances, how to compute the values of others? In other words, how to define functions in a language, which return the satisfaction value of a Fragment or relation instance, and take into account already known satisfaction values of other Fragments and relation instances? (Section 9.1.2)
- How does the presence of Alternatives and Combinations in a model (and so, of r.xor in a language) influence the computation of satisfaction values in a language? (Sections 9.1.3 and 9.1.4)

I use satisfaction values because I discussed satisfaction already in relation to influence relations. However, the discussion in this section remains relevant for any binary Value Type. As for how to compute values, I use a simple approach often informally referred to as "value propagation", where a relation instance from y to x is seen, roughly speaking, as a pipe that conducts a value from y to x, whereby the value to conduct depends on the value of y and on the specifics of the relation. Values thus get pushed through potentially many such pipes to a Fragment, and there is then a rule which aggregates them, and outputs a single value for that Fragment. There are other ways to compute values on model parts. I will mention some of them, and leave others outside the scope of this tutorial.

9.1.1 Binary Value Type

To motivate the use of binary Value Types, recall the first condition in the DRP. It says that there has to be a proof of requirements from domain knowledge and specifications. The more general idea is this: It should be shown that if conditions that domain knowledge and specifications describe are satisfied, then the conditions described with requirements are satisfied as well. This gives the following Language Service.

Language Service

SatReq: Are all requirements satisfied in the model M?

To deliver s.SatReq, it is necessary to have a Value Type for satisfaction. Given how s.SatReq is phrased, it looks enough to have two values, for satisfied and not satisfied. If s.SatReq asked, instead, for how well requirements were satisfied, then a binary Value Type would not work.

To deliver, then, s.SatReq, I will use v.Satisfaction, a binary Value Type such that

$$\text{v.Satisfaction} = \{1, 0\}$$

where 1 reads "satisfied" and 0 reads "not satisfied".

s.SatReq mentions requirements, so that the language has to distinguish requirements Fragments from others. I will keep using the three categories defined earlier, namely c.r, c.k, and c.s.

What, in a model, gets a value of v.Satisfaction? A variable, which is associated to every Fragment and every relation instance. The language thus also needs a set of variables. There will be as many variables as there are Fragments and relation instances. As I am working with a single Value Type here, all variables will take values from v.Satisfaction.

9.1.2 Value Propagation

The language needs to represent that the satisfaction value a Fragment depends on the satisfaction values on one or more other Fragments, and if it does, then how exactly. This is done by having a function which is sensitive to the relations between Fragments. Given the motivation discussed earlier for influence relations, the language will include r.inf.pos and r.inf.neg. It will also need another function, which is presented in Sections 9.1.3 and 9.1.4.

Exercise 20: Define functions that assign satisfaction values across positive and negative influence relations

> Suppose that there is a positive influence relation instance $(y, x) \in$ r.inf.pos. Define a function which returns the satisfaction value of x, when the satisfaction value of y is known. Do the same for $(y, x) \in$ r.inf.neg.

Recall that influence relations were not defined specifically with v.Satisfaction in mind, but simply to represent, when it exists, the information that satisfaction of a Fragment depends on that of another. The next language design decision to make, then, is to define how exactly the satisfaction value of a Fragment influences that of another, when there is an influence relation between them. The following rules come to mind, for $(y, x) \in$ r.inf.pos in a model M:

- if y gets the value 1 from v.Satisfaction, x should get 1 as well, if one ignores all (if any) other influence relation that may be targeting x in M,
- if y gets 0, then x gets 0, too, if one ignores all (if any) other influence relation that may be targeting x in M.

I emphasised in both rules above that they are local: they say which value to assign to x only by considering the value that y has, and that the relation instance is a positive influence (rather than a negative influence). The rules ignore all other positive or negative influences to x, from Fragments other than y.

To have these rules in a language, I add a new function, f.sat.inf.pos, which relies on f.sat to return the satisfaction value of a Fragment. f.sat remains undefined for the moment. When I define it later (one in Section 9.1.3, another in Section 9.1.4), it will say what the satisfaction value of a Fragment is, given potentially many positive and negative influence relation instances to that Fragment. This is different than f.sat.inf.pos, which concentrates on the satisfaction value of a single positive influence relation instance.

I will write $\langle x, t, v \rangle$ for a variable of v.t which is associated to the Fragment or relation instance x, and whose value is v. This is called a "value assignment".

Function

Positive influence satisfaction
(f.sat.inf.pos)

Input

$(y, x) \in$ r.inf.pos and model M.

Do

$v = 1$ if y is satisfied in M, and $v = 0$ otherwise.

Output

$\langle (y, x), \text{v.Satisfaction}, v \rangle$.

Language Services

- **s.WhPosInfSat**: What is the v.Satisfaction value of $(y, x) \in$ r.inf.pos? : It is the value assignment $\langle (y, x), \text{v.Satisfaction}, v \rangle$.

The function f.sat.inf.pos is based on the idea of "propagating" values. To see what this amounts to, suppose that there are Fragments y and x in a model M, and there is positive influence from y to x. So if y is satisfied, then this positively influences the satisfaction of x. But you cannot simply conclude that x is in fact satisfied, because there may be other influences, positive or negative, which target x, from Fragments other than y.

Propagation consists of seeing relation instances as a kind of pipes, each of which propagates a value to its target. There may be many relation instances which propagate different values to the same target, and therefore, it is necessary (as I will discuss in Sections 9.1.3 and 9.1.4) to have rules which aggregate all these values that a Fragment receives, and concludes one satisfaction value for that Fragment.

When valuation involves value propagation, the values on relation instances may be somewhat confusing, as in the function below. It propagates satisfaction values of negative influence.

Function

Negative influence satisfaction
(f.sat.inf.neg)

Input

$(y, x) \in$ r.inf.neg and model M.

Do

If y is not satisfied in M, then x should be, and $v = 1$. If y is satisfied in M, then x should not, and so $v = 0$.

Output

$\langle (y, x), \text{v.Satisfaction}, v \rangle$.

Language Services

- **s.WhNegInfSat**: What is the v.Satisfaction value of $(y, x) \in$ r.inf.neg? : It is the value assignment $\langle (y, x), \text{v.Satisfaction}, v \rangle$.

A satisfied negative influence is thus not an influence which successfully negatively affects its target, but one which fails to do so, and therefore propagates 1 to x in f.inf.neg.

The next step is to define f.sat which computes the satisfaction value of a Fragment, based on all positive and negative influences to that Fragment.

For some Fragments, the satisfaction value will be computed, for others manually assigned. So I need rules for how to compute values, as I otherwise cannot answer such questions as "What should be the v.Satisfaction value of a Fragment x, when x is the target of two or more positive and/or negative influence relations?" For example, what is the satisfaction value of x, if f.sat.inf.pos$(y, x) = 1$, f.sat.inf.pos$(z, x) = 0$, and f.sat.inf.neg]$(w, x) = 0$?

I distinguish two cases below, when the language cannot represent Alternatives in Section 9.1.3, and when it can in Section 9.1.4. This allows me to illustrate the difference that the representation of Alternatives makes on valuation with v.Satisfaction.

9.1.3 Without Alternatives

Suppose that the language cannot represent Alternatives and Combinations. It has no r.xor. Let $(p_1, x), \ldots, (p_n, x)$ be instances of r.inf.pos and $(q_1, x), \ldots, (q_m, x)$ be instances of r.inf.neg, all targeting the Fragment x. Consider the following rules:

1. if for all $i = 1, \ldots, n$, it is the case that f.sat.inf.pos$((p_i, x)) = 1$, and for all $j = 1, \ldots, m$, f.sat.inf.neg$((q_j, x)) = 1$, then the satisfaction value of x is 1,

2. in all other cases, the satisfaction value of x is 0.

I can add these rules to a language via the function f.sat.

Function

Satisfaction
(f.sat)

Input

Fragment x and model M.

Do

Let $\{(p_1,x),\ldots,(p_n,x)\} \subseteq$ r.inf.pos be the set of all positive influence relation instances to x in M, and $\{(q_1,x),\ldots,(q_m,x)\} \subseteq$ r.inf.neg be the set of all negative influence relation instances to x in M. Then,

$$v = \prod_{i=1}^{n} \text{f.sat.inf.pos}((p_i,x),M) \cdot \prod_{j=1}^{m} \text{f.sat.inf.neg}((q_j,x),M)$$

Above, f.sat.inf.pos$((p_i,x),M)$ returns the satisfaction value of (or propagated by) $(p_i,x) \in$ r.inf.pos in M, and f.sat.inf.neg$((q_j,x),M)$ returns the satisfaction value of $(q_j,x) \in$ r.inf.neg in M.

Output

$\langle x,$ v.Satisfaction, $v\rangle$

Language Services

- **s.WhSat**: What is the satisfaction value of x in M? : It is $\langle x,$ v.Satisfaction, $v\rangle$.

Which rules are relevant for f.sat depends on what exactly these rules should do for you. Above, the rules reflect the idea that x will be satisfied only if everything influencing it positively is satisfied as well, and everything influencing it negatively is not satisfied. In some sense, it reflects a demanding and defensive attitude about when Fragments are satisfied. If any one of these two conditions fails, for example, a Fragment is satisfied, and negatively influences x, it will not matter that there may be other Fragments which are satisfied and positively influence x. The conclusion will be that x is not satisfied.

Exercise 21: Define a function which assigns a satisfaction value to all Fragments which are not influenced

Take a language which has Fragments and positive and negative influence relations, and can assign v.Satisfaction values. Define a function which assigns a satisfaction value to every Fragment which is not influenced in a model of that language, that is, every Fragment which is not a target of a positive or negative influence relation.

To give a satisfaction value of some x, f.sat needs all influence relations to x. But what if there are none? f.sat cannot assign a satisfaction value to x, and neither can f.sat.inf.pos and f.sat.inf.neg. You need to choose in another way the values of Fragments, whose values cannot be computed with these three functions.

In addition to the three functions, another function is needed to assign satisfaction values for every Fragment which is target of no influence relation. These are Fragments from which you start propagating satisfaction values. If you think in terms of graphs over influence relations, then this amounts to assigning a satisfaction value to every leaf node *only*, and then using the three functions mentioned above, to compute the satisfaction values of other Fragments. This leads to f.sat.leaf below, which takes a Fragment with no influence relations and assigns a satisfaction value to it.

Function

Assume a satisfaction value for a non-influenced Fragment
(f.sat.leaf)

Input

Fragment x and model M, such that there is no $(y,x) \in$ r.inf, $(y,x) \in$ r.inf.pos, and $(y,x) \in$ r.inf.neg in M with y also in M.

Do

If you assume that x is satisfied, then $v = 1$, else if you assume that x is not satisfied, then $v = 0$, else leave v without value.

Output

$\langle x$, v.Satisfaction, $v\rangle$

Language Services

- **s.WhAsmSatLf**: Which, if any, is the assumed v.Satisfaction value of x in M? : $\langle x$, v.Satisfaction, $v\rangle$ if $v \in \{0,1\}$, otherwise no v.Satisfaction value is assumed for x.

The four functions, f.sat.inf.pos, f.sat.inf.neg, f.sat, and f.sat.leaf are enough to assume and compute satisfaction values on models that relate Fragments with positive and negative influence relations. The following language puts these notions together.

Language

Rigel

Language Modules

F, **T**, **V**, r.inf.pos, r.inf.neg, f.brel2g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat, f.sat.leaf

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, F = c.r∪c.k∪c.s and c.r∩c.k∩c.s = ∅. Influences are over Fragments, r.inf.pos ⊆ F × F, r.inf.neg ⊆ F × F. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\mathbf{V} \subseteq (\mathsf{F} \cup \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg}) \times \mathbf{T} \times \mathsf{v.Satisfaction}.$$

The language has one binary Value Type, **T** = {v.Satisfaction}, and v.Satisfaction = {1,0}.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & r \mid k \mid s \\ \gamma & ::= & \beta(\alpha) \\ \delta & ::= & (\gamma, \gamma) \\ \epsilon & ::= & \langle \alpha, \zeta, \eta \rangle \\ \phi & ::= & \gamma \mid \delta \mid \epsilon \end{array}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in \mathsf{F}$. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. δ symbols denote positive and negative influence relations. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in \mathbf{T}$. η denotes a value of a Value Type, and as there is one Value Type, then $\mathscr{D}(\eta) \in$ v.Satisfaction. ϵ symbols denote value assignments, $\mathscr{D}(\epsilon) \in \mathbf{V}$.

Language Services

Those of relations and functions in the language, and s.SatReq.

While L.Rigel does not have r.xor, and therefore no Alternatives and Combinations, its models can represent mutually exclusive value assignments on same Fragments and relation instances. This is useful, because, for example, different people may use f.sat.leaf, and they may have different assumptions about the values of leaf Fragments. Or the model user wishes to ask what-if kinds of questions, such as "What if all leaf Fragments get these satisfaction values, as opposed to these other satisfaction values?" and wishes to compare the Outcomes (more on this in Section 11). This is illustrated in Example 9.1 below.

An *Outcome* is the assignment of a single value per Value Type, to all Fragments and relation instances in a model. In Example 9.1, Figures 16(a) and 16(b) do not

include Outcomes. Figure 16(c) includes one Outcome, and Figure 16(d) includes two. An Outcome can be specific to one Value Type, as in Figures 16(c) and 16(d), where only v.Satisfaction values can be assigned anyway, due to the specifics of the language used. When I want to say that an Outcome has values of only one, or some specific set of Value Types, I will write so. For example, Figures 16(c) and 16(d) show v.Satisfaction Outcomes.

Example 9.1. This example illustrates how L.Rigel computes value assignments in a model. Figure 16 shows four models in L.Rigel.

The first model in Figure 16(a) shows a model with assignments of satisfaction values to Fragments with no incoming positive or negative influence relations. This assignment is a result of applying f.sat.leaf. There can be other assignments, as the values depend entirely on the model user who is assigning them.

The second model, in Figure 16(b) is the result of applying f.sat.inf.pos andf.sat.inf.neg on positive and negative influence relation instances which are directly connected to the leaf Fragments. You can think of this model as showing one step of propagating the satisfaction values assumed and shown in the first model in Figure 16(a).

The third model shows the satisfaction values assigned after applying f.sat.inf.pos, f.sat.inf.neg, and f.sat to all influence relation instances and Fragments in the model.

The model in Figure 16(d) shows two Outcomes, that is, two assignments of satisfaction values to every Fragment and relation instance. Values for one Outcome are shown on black squares, and on grey squares for the other. •

L.Rigel delivers s.SatReq in the following way. Given a model, you apply f.sat.leaf and assign one satisfaction value to every leaf Fragment. You then propagate satisfaction values using f.sat.inf.pos, f.sat.inf.neg, and f.sat, until you have one Outcome. If that Outcome assigns the satisfaction value 1 to every requirement in the model, then the answer to s.SatReq is affirmative, and is "no" otherwise.

9.1.4 With Alternatives

Exercise 22: Define a language which can propagate satisfaction values over Alternatives and Combinations

How would you define a language which can represent the same as L.Mirfak, but also lets you have Alternatives and Combinations, and assign v.Satisfaction values on Fragments and relation instances?

If you add f.sat.inf.pos, f.sat.inf.neg, f.sat, and f.sat.leaf to L.Mirfak, this will not produce appropriate satisfaction values. This is because f.sat ignores r.xor. For illustration, consider this question: what is the satisfaction value of x, if there are only two positive influence relation instances to x, namely, $(y,x) \in$ r.inf.pos and $(z,x) \in$ r.inf.pos, and if they are such that

$$((y,x),(z,x)) \in \text{r.xor},$$
$$\text{f.sat.inf.pos}((y,x),M) = 1,$$
$$\text{f.sat.inf.pos}((z,x),M) = 0.$$

Clearly, f.sat will say that $\langle x, \text{v.Satisfaction}, 0\rangle$, but this is not correct. The correct value assignment is $\langle x, \text{v.Satisfaction}, 1\rangle$, because (y,x) and (z,x) are Combinations.

The language needs a replacement for f.sat, which is sensitive to the presence of r.xor instances. This replacement is f.sat.x, defined below.

Function

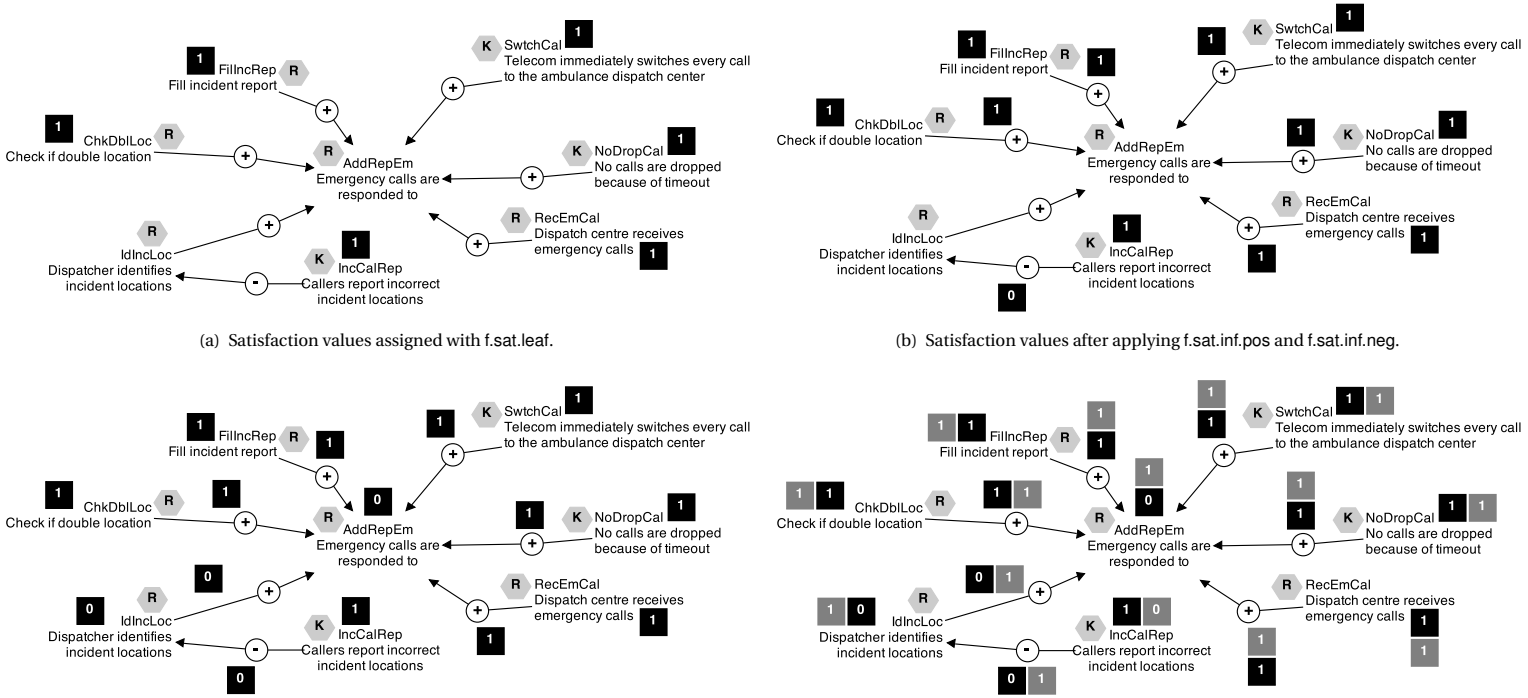
Satisfaction over Alternatives
(f.sat.x)

Input

Fragment x and model M.

Do

1. Find the set $\{(p_1,x),\ldots,(p_n,x)\} \subseteq$ r.inf.pos of all positive influence relation instances to x in M.
2. Find the set $\{(q_1,x),\ldots,(q_m,x)\} \subseteq$ r.inf.neg of all negative influence relation instances to x in M.
3. Find the set $\{a_1,\ldots,a_h\} \subseteq$ r.xor of all r.xor instances over the positive and negative influence relation instances to x in M, that is, over $\{(p_1,x),\ldots,(p_n,x)\} \cup \{(q_1,x),\ldots,(q_m,x)\}$.
4. Let $M_x = (X,R,A)$ such that $X = \{x\}$, $R = \{(p_1,x),\ldots,(p_n,x)\} \cup \{(q_1,x),\ldots,(q_m,x)\}$, and $A = \{a_1,\ldots,a_h\}$.
5. Find the set O of all Combinations in M_x by applying f.find.all.cb on M_x. Note that O includes all Combinations of positive and negative influence relations to x in M.



(a) Satisfaction values assigned with f.sat.leaf.

(b) Satisfaction values after applying f.sat.inf.pos and f.sat.inf.neg.

(c) Satisfaction values of all Fragments and relation instances in the model.

(d) Two sets of assumed satisfaction values and their propagation.

Figure 16: Models and value assignments in L.Rigel.

6. Suppose that there are g Combinations in O, so that $O = \{O_1, \ldots, O_g\}$. Suppose that a generic Combination O_k includes the Choices $\{p_b, \ldots, q_c, \ldots\}$. For each Combination O_k, compute

$$v_{O_k} = \prod_{(p_i,x)\in O_k} \text{f.sat.inf.pos}((p_i, x), M) \cdot \prod_{(q_j,x)\in O_k} \text{f.sat.inf.neg}((q_j, x), M)$$

7. Let $max(v_{O_k})$ be the maximal value among all values $v_{O_1}, \ldots, v_{O_g}$.
8. Satisfaction value of x in M is given by $\langle x, \text{v.Satisfaction}, max(v_{O_k})\rangle$.

Output

$\langle x, \text{v.Satisfaction}, max(v_{O_k})\rangle$.

Language Services

- s.WhSat: $\langle x, \text{v.Satisfaction}, max(v_{O_k})\rangle$.

Instead of taking the product of all satisfaction values, on all edges to x in a model, as f.sat did, f.sat.x first needs to find all allowed combinations of influences to x, and then compute the product of satisfaction values for each of these. An allowed combination of influences includes a subset of all influences to x, whereby that subset has to include all influences which are not mutually exclusive. To illustrate how f.sat.x works, I start by defining a language which uses it.

Language

Capella

Language Modules

F, **T**, **V**, r.inf.pos, r.inf.neg, r.xor, f.brel2g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat.x, f.sat.leaf

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathsf{F} = \text{c.r} \cup \text{c.k} \cup \text{c.s}$ and $\text{c.r} \cap \text{c.k} \cap \text{c.s} = \emptyset$. Influences are over Fragments, $\text{r.inf.pos} \subseteq \mathsf{F} \times \mathsf{F}$, $\text{r.inf.neg} \subseteq \mathsf{F} \times \mathsf{F}$. r.xor instances are over influence relations of the same type,

$$\text{r.xor} \subseteq (\text{r.inf.pos}^n) \cup \text{r.inf.neg}^n)$$

Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\mathbf{V} \subseteq (\mathsf{F} \cup \text{r.inf.pos} \cup \text{r.inf.neg}) \times \mathbf{T} \times \text{v.Satisfaction}.$$

The language has one binary Value Type, $\mathbf{T} = \{\text{v.Satisfaction}\}$, and $\text{v.Satisfaction} = \{1, 0\}$.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{aligned}
\alpha &::= x \mid y \mid z \mid \ldots \\
\beta &::= r \mid k \mid s \\
\gamma &::= \beta(\alpha) \\
\delta &::= (\gamma, \gamma) \\
\epsilon &::= (\delta, \delta, \ldots) \\
\theta &::= \langle \alpha, \zeta, \eta \rangle \\
\phi &::= \gamma \mid \delta \mid \epsilon \mid \theta
\end{aligned}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in \mathsf{F}$. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in \text{c.r}$, $\mathscr{D}(k(\alpha)) \in \text{c.k}$, $\mathscr{D}(s(\alpha)) \in \text{c.s}$. δ symbols denote positive and negative influence relations. ϵ symbols denote r.xor instances, $\mathscr{D}(\epsilon) \in \text{r.xor}$. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in \mathbf{T}$. η denotes a value of a Value Type, and as there is one Value Type, then $\mathscr{D}(\eta) \in \text{v.Satisfaction}$. θ symbols denote value assignments, $\mathscr{D}(\theta) \in \mathbf{V}$.

Language Services

Those of relations and functions in the language, and s.SatReq.

Example 9.2. Figure 17 illustrates L.Capella and the assignment of satisfaction values when there are r.xor instances in a model.

Figure 17(a) shows a model in L.Capella. The model has no value assignments. It shows four r.xor instances. The Alternatives shown give two Combinations. One Combination is shown in Figure 17(b), where the removed Alternatives are in bold. The other Combination is in Figure 17(d), and again, removed Alternatives are in bold.

Suppose that you apply f.sat.leaf to the leaf Fragments in the first Combination, and that you get the values shown in Figure 17(b). If you then propagate these values to ChoAmb, you get the satisfaction values shown in Figure 17(c). In other words, if you actually deleted the bold relations in Figure 17(c), the satisfaction value of ChoAmb would be 1.

Consider now the second Combination. Suppose that you applied f.sat.leaf to the leaf Fragments in that Combination, and that you got the values shown in Figure 17(d). If you propagate these values, the satisfaction value of ChoAmb is 0.

Let O_1 denote the first, and O_2 the second Combination. The conclusion from the above is that $v_{O_1} = 1$ and $v_{O_2} = 0$. Following f.sat.x, the result is

$$\langle \text{ChoAmb}, \text{v.Satisfaction}, 1 \rangle.$$

The result reflects the idea that there are two Combinations for ChoAmb, and that ChoAmb should be satisfied, if at least one of these Combinations propagates the satisfaction value 1 to ChoAmb. Because Combinations are mutually exclusive, there is no need for more than one Combination to propagate the satisfaction value 1 to ChoAmb.

Figure 17(f) shows another set of leaf value assignments, shown on grey squares, and the propagation of these values to ChoAmb. The difference is that in that Figure, grey squares carry value assignments where the first Combination propagates the satisfaction value 0 to ChoAmb, while the second Combination now propagates the satisfaction value 1. •

9.2 How to Combine Several Binary Value Types?

This section looks at how to have more than one Value Type in a language. It focuses on a simple case when there are two binary Value Types. Consider the following Language Service.

Language Service

AppSat: Which requirements in the model M are both approved by all stakeholders, and satisfied?

The language needs two Value Types, one for satisfaction and the other for approval. They will be called v.Satisfaction and v.Approval. If you allow a stakeholder to either approve or not a requirement, then v.Approval is a binary Value Type. By analogy to v.Satisfaction, which remains here the same as in Section 9.1, there is v.Approval = $\{1, 0\}$, where 1 reads "approved", and 0 "not approved".

Then, it is necessary to decide how the approval of a Fragment depends on the approval of other Fragments, if in any way. One option is to ask stakeholders to assign an approval value to each requirement Fragment, and therefore *not* compute the approval value of requirements. Another is to allow influence relations (or some other relations in the language) to be significant for approval, perhaps in the same way that they were significant for satisfaction in Section 9.1. That is, if there is $(y, x) \in$ r.inf.pos, and it is known that y is approved, than a rule would say how this should be taken into account to compute the approval value of x.

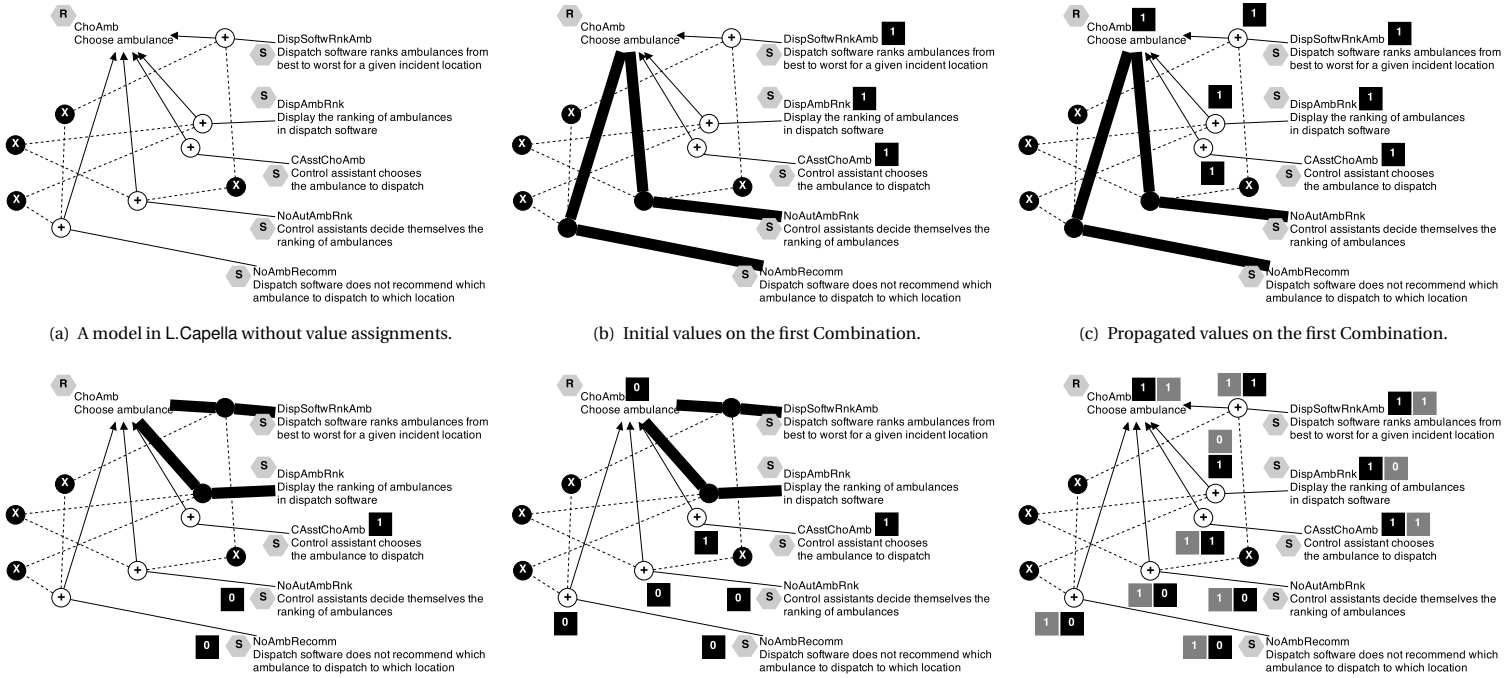
Section 9.2.1 looks at the case where the approval values are assigned to every Fragment manually, so that there is no need for rules to compute those values. Section 9.2.2 focuses on the case where missing approval values can be computed from those that exist in a model.

9.2.1 When Value Assignments are Independent

Consider the following exercise.

Exercise 23: Define a function that assigns an independent approval value to a Fragment

> Suppose that the approval value of a Fragment or relation instance is independent from the approval value of another Fragment, or of a relation instance. Moreover, suppose that the satisfaction values are independent from approval values, and *vice versa*. If I approved Fragment x, then this has nothing to do with whether I will approve Fragment y, or whether y is satisfied. How would you enable a language to assign approval values in this way and deliver s.AppSat?



(a) A model in L.Capella without value assignments.

(b) Initial values on the first Combination.

(c) Propagated values on the first Combination.

(d) Initial values on the second Combination.

(e) Propagated values on the second Combination.

(f) Two v.Satisfaction assignments to a model in L.Capella.

Figure 17: Models and v.Satisfaction value assignments in L.Capella.

You can add v.Approval to L.Rigel, and add a function for asserting approval values which works in the same way as f.sat.leaf. The function is as follows.

Function

Assume independent approval value
(f.app.asg.ind)

Input

Fragment x.

Do

If a stakeholder approves x, then $v = 1$, else $v = 0$.

Output

$\langle x, \text{v.Approval}, v \rangle$

Language Services

- **s.AsmApp**: Is x approved by a stakeholder? : Yes, if $\langle x, \text{v.Approval}, 1 \rangle$, no otherwise.

Assigning approval values in a model M with f.app.asg.ind consists of asking a stakeholder to approve each Fragment and relation instance.

Two issues arise:

1. There can be many stakeholders, so you should decide if the approval value reflects the approval of a single stakeholder, of some, or of all. The issue is whether to allow the assignment of tuples of approval values to model parts, with one approval value per stakeholder. f.app.asg.ind assigns individual values.

2. How to read and use, if in any way, the combination of a satisfaction and approval value on a Fragment? For example, is there some new information to conclude from knowing both that a Fragment is satisfied and that it is not approved? Satisfaction and approval values still are independent, but the question is if you should draw some additional conclusion from knowing both the satisfaction and approval value of a Fragment or relation instance.

On the first issue, if the models need to show all approval values, from all stakeholders, then the language should allow every model part to carry as many approval values as there are stakeholders. The approval value of a model part would be a tuple, each element being the approval value of one stakeholder.

If it is necessary to decide a single approval value of a model part, when there are many approval values coming from many stakeholders, then the language needs to have rules for aggregating approval values. For example, aggregation rules can be that if all stakeholders approve a model part, then it is approved, or that if the majority approves a model part, then it is approved, and so on. Research in group decision making [30] and social choice [26, 4] are one source of such aggregation rules.

The second issue is if knowing both the satisfaction and approval values *together* gives some additional information for problem solving, and which is useful for deciding what to do next with the model. For example, if a model part is both satisfied and approved, then it is probably more interesting to look at other model parts in the next steps of problem-solving. If model parts are seen as representations of parts of the problem being solved and of its potential solutions, then a satisfied and approved model part can be considered as a solved problem part.

In this same line of thinking, if a model part is not satisfied, but is approved, then it will need to be solved, that is, it is necessary to change the model in such a way as to ensure that, in the changed model, the model part is both satisfied and approved. There is the case of a satisfied and not approved part, which can become solved by, for example, negotiating its approval with or among stakeholders, or by removing from the model those parts which satisfy it, yet are unnecessary for satisfying the approved model parts. The final case is that of a part which is neither satisfied, nor approved. It may thereby not even be a part of the problem, even if it is part of the model. Table 1 summarises these ideas.

Table 1: Combinations of v.Satisfaction and v.Approval values.

	Approved	*Not approved*
Satisfied	No action needed	Negotiate or remove
Not satisfied	Find a way to satisfy it	Ignore

The more general point for language design is that allowing two or more Value Types raises the question of how to use the various combinations of these values in problem solving, if to use them at all.

If the value combinations are useful, then this can be captured by a new Value Type, and functions for assigning and, or computing their values. For illustration, two new Value Types are defined blow, one from combinations of v.Approval only, the other from both v.Approval and v.Satisfaction.

Example 9.3. v.MajApp = {1,0} is such that 1 is given to a model part if half or more of all stakeholders have assigned the v.Approval value 1 to this model part. This gives the following function. •

Function
Majority approval **(f.app.maj)**
Input Fragment x.
Do If more than half of all stakeholders approve x, then $v = 1$, else $v = 0$.
Output $\langle x, \text{v.MajApp}, v \rangle$.
Language Services • **s.IsMajApp**: Is x approved by the majority of stakeholders? : Yes, if $\langle x, \text{v.MajApp}, 1 \rangle$, no otherwise.

Example 9.4. v.SatNext = {1,0} is used to mark model parts which are approved and not satisfied. As they are approved, there is no need to discuss them further with stakeholders, but focus on how to change the model to satisfy them. These values are assigned with the following function. •

Function
Satisfy next **(f.sat.nxt)**
Input Fragment x.
Do $v = 1$ if $\langle x, \text{v.Satisfaction}, 0 \rangle$ and $\langle x, \text{v.Approval}, 1 \rangle$, else $v = 0$.
Output $\langle x, \text{v.SatNext}, v \rangle$.
Language Services • **s.DoSatNext**: Should problem solving focus next on how to satisfy x? : Yes, if $\langle x, \text{v.SatNext}, 1 \rangle$.

Neither v.MajApp, nor v.SatNext are defined over all four possible combinations of v.Satisfaction and v.Approval values. This is because a Value Type which is defined over all four combinations is not binary, but instead an unordered set of four values. It is discussed in Section 9.3.

9.2.2 When Value Assignments are not Independent

In Section 9.2.1, only f.app.maj computed the approval value of a Fragment from other approval values on that same Fragment. There were no rules about how, for example, to compute the approval value of x from those of other Fragments, which x is somehow related to.

Exercise 24: Define new relations, functions, or otherwise, for propagating approval values

If you have a language which cannot assign v.Approval values, how

would you change that language so that it can assign these values to Fragments and relation instances, along similar lines as f.sat.inf.pos, f.sat.inf.neg, f.sa or f.sat.x, and f.sat.leaf did for v.Satisfaction values? Would you need new relations in that language? Which new functions would you add, and why?

To compute approval values in models, rather than assign them manually to all Fragments, you need to make analogous decisions to those made for functions which computed satisfaction values in Section 9.1. Therefore, if the language does not represent alternatives, and you want to assign approval values by propagating them, then you need to make the following decisions:

1. What is the relation r whose instance $(y,x) \in r$ should exist, in order for the approval value of the Fragment x to depend on the approval value of the Fragment y?
2. If there is a relation instance $(y,x) \in r$, and the approval value of y is 1 (or 0), what should be the approval value of x?
3. If there are several relation instances $(y_1,x) \in r_1, \ldots, (y_n,x) \in r_n$, and approval values of $y_1, \ldots, y_n$ are not the same, then what should be the approval value of x?
4. If there are no r relation instances to x, then what should be the approval value of x?

If the language can represent alternatives with r.xor, then it is necessary to make sure the function which computes approval values takes into account the r.xor instances. This is by analogy to f.sat.x which is sensitive to r.xor, and f.sat which is not.

Recall how the questions above were answered for v.Satisfaction. The presence of r.inf.pos or r.inf.neg between two Fragments x and y meant that the satisfaction value of one depended on that of the other. If you think in terms of value propagation, positive and negative influence relations were used to propagate satisfaction values. That answers the first question. f.sat.inf.pos and f.sat.inf.neg defined how satisfaction value of x depends on that of y, in case when there is, respectively, $(y,x) \in$ r.inf.pos or $(y,x) \in$ r.inf.neg. For languages which do not represent alternatives, f.sat answers to the third question above; when there are alternatives in models, f.sats answers the third question. Finally, f.sat.leaf was the answer to the fourth question.

9.3 What If a Value Type Is a Set of Values?

What if a Value Type is a set of values, and there is no order over them? In Section 9.2, there were four combinations of binary values from two core binary Value Types, v.Satisfaction and v.Approval. Table 1 gave a reading of these combinations. The four combinations can be used to define the three values of a new Value Type. This new Value Type is called v.ToDo. These values are as follows:

- *Done*, when satisfaction and approval values are both 1,
- *Operationalise*, when satisfaction is 0 and approval 1,
- *Negotiate or remove*, when approval is 0, regardless of satisfaction.

Each value suggests what to do next about the Fragment or relation instance it is assigned to, hence the name of the Value Type. The rules for assigning this Value Type are straightforward, as its values are fully determined by the satisfaction and approval values. I leave it to the reader as an exercise to define these rules in a function, and to add that function to any of the languages defined so far in Section 9.

To illustrate a more complicated Value Type which is also a set of unordered values, recall that I defined five questions, *Who*, *How*, *When*, *Where*, and *WhoFor* and the corresponding unary relations. Suppose that you want to make a language which delivers the following Language Service.

Language Service

WhichDetail: Which of the questions among *Who*, *How*, *When*, *Where*, and *WhoFor* were not asked for the Fragment x?

There are different ways to deliver s.WhichDetail, but I will focus on one which uses a Value Type, whose values are assigned exclusively to Fragments. The assigned value is such that it tells the modeller exactly those questions which were not asked for that Fragment. Examples of its values are the set {*When*, *Where*, *WhoFor*} when these three questions are not answered for a Fragment, or{*Who*} if only that question was not answered for the Fragment.

This new Value Type is v.AskNext, and it has 2^5 possible values. The value to assign to a Fragment x is computed using simple rules, which look at the presence or absence of r.q instances that target x, where q is any of the five questions. The following function defines these rules.

Function

What to ask next
(f.ask.next)

Input

Fragment x and model M.

Do

Let V be an empty set. Let $I_x = \{(p_1, x), \ldots, (p_n, x)\} \subseteq$ r.ifm be the set of all instances of r.ifm in M which end in x. For each $q \in \{Who, How, When, Where, WhoFor\}$, if there is $(p_i, x) \in I_x$ such that $(p_i, x) \in$ r.q, then add q to V.

Output

$\langle x,$ v.AskNext, $V \rangle$.

Language Services

- s.WhichDetail: Those which are not in V, in $\langle x,$ v.AskNext, $V \rangle$.

An important idea illustrated with all Value Types so far, and in particular with v.SatNext, v.ToDo, and v.AskNext, is that values on model parts can act as cues for what to do next with the model, and more generally, what next steps to take in problem solving.

9.4 What If Some Values Cannot Be Assigned After Others?

What if some sequences of assignments of values to the same Fragment or relation instance are not allowed? That is, you can assign some value v_1 only to those Fragments or relation instances which are already assigned the value v_2, and not some other value. Suppose that the aim is to design a language which delivers the following Language Service:

Language Service

WorkProgrRep: What is the progress in the implementation of the specifications in the model M?

This Language Service can be interesting for teams where the model is used to distinguish specifications which are implemented, from those that remain to be implemented. If the model includes requirements, domain knowledge, and specifications, asking about the progress of work may refer to how close the team is to finding a solution such that the requirements are satisfied. Or if a solution was found, if, or what parts of it, are implemented, and thereby get an idea about how much of the system is already in place. These two are two different ways to understand "work progress". I will focus on the second one, because the first was discussed earlier, with v.SatNext, v.ToDo, and v.AskNext.

Suppose that the team is using the following simple steps for each specification Fragment x:

1. check if specification x is approved by the system designer, and if yes then
2. check if there is an estimate of time required to implement x, and if yes then
3. check if x is added to product roadmap, and if yes then
4. check if x is ready for testing, and if yes
5. check if x is approved for release, and if yes, then stop.

Exercise 25: Define a Value Type whose values can be assigned in a specific sequence

> Define a new Value Type which has, as its values, the names of steps in the process outlined above. How do you ensure, in a language which has that Value Type, that its values are assigned in the appropriate sequence? For example, the value for the third step in the process cannot be assigned to the same Fragment if that Fragment carries the first value.

The process suggests values for a new Value Type. Call it v.ProgrStatus. Let it have the following values, each corresponding to the respective step above: *DesignApproved*, *EstimateDone*, *InRoadmap*, *TestReady*, and *ApprovedForRelease*. Assuming that these values are manually assigned in a model (rather than computed). Given the discussions of valuation so far, it should be clear how to add this Value Type to any of the languages in the preceding sections.

What I want to emphasise with this Value Type, is that its values alone do not convey the idea which is informally clear in s.WorkProgrRep and from the steps described above, namely, that there is an order, from the approval that x should be done, or implemented, or otherwise completed, to its completion and release.

The order introduces constraints on when a value can be assigned to x, and depends on the value which x already has. Suppose that I want to force modellers to assign the values of v.ProgrStatus according to this order. That is, if some x is assigned *DesignApproved*, then it cannot be assigned *InRoadmap*. The modeller can change the value on x from *DesignApproved* to *EstimateDone*, and only then change the value to *InRoadmap*, not go straight from *DesignApproved* to *InRoadmap*. This can be done with a function which checks if the assignment of a value of v.ProgrStatus satisfies the order over the values. The function is as follows.

Function

Check progress status sequence
(f.chk.progrstatus)

Input

Fragment x and two value assignments $\langle x, \text{v.ProgrStatus}, v_{old}\rangle$ and $\langle x, \text{v.ProgrStatus}, v_{new}\rangle$, where $\langle x, \text{v.ProgrStatus}, v_{new}\rangle$ is the new value that a modeller wishes to add to x, to replace $\langle x, \text{v.ProgrStatus}, v_{old}\rangle$.

Do

Check if (v_{old}, v_{new}) is in the following set

$$\{ \quad (none, DesignApproved), (DesignApproved, EstimateDone), (EstimateDone, InRoadmap), (InRoadmap, TestReady), (TestReady, ApprovedForRelease)\}$$

If yes, then let $v = 1$, else $v = 0$.

Output

v.

Language Services

- **s.ProgrStatusOk**: Can $\langle x, \text{v.ProgrStatus}, v_{new}\rangle$ replace $\langle x, \text{v.ProgrStatus}, v_{old}\rangle$? : Yes, if $v = 1$, otherwise no.

The function takes the current (old) assignment of a v.ProgrStatus value, and checks if the new value assignment, which replaces the old, satisfies the constraints on the sequence in which the values of this Value Type can be assigned. Returning to s.WorkProgrRep, notice that it is delivered as soon as it is possible to assign values of v.ProgrStatus to model parts in a language.

9.5 What If a Value Type Is Over Reals?

The hypothetical work process in Section 9.4 has a step, when one checks if there exists an estimate of time required to implement what a Fragment describes. If these estimates need to be recorded in models, then there can be a new Value Type, call it v.ImplTime, whose allowed values are positive reals, integers most likely.

Depending on the specifics of the language which has this Value Type, the assignment of implementation time values can be entirely manual or partly automated. In absence of automation, the language would require that an individual, or more of them, assign a positive integer value to each Fragment.

In the partly automated case, values assigned to some Fragments would be used to compute values on others. Let the language have positive and negative influence relations, for example. Suppose that there are only two positive influence relations (y, x) and (z, x) to a Fragment x. If the assigned implementation time to y is 10 man-hours, and to z is 5 man-hours, the language could include a function which sums these two, and returns 15 man-hours a the implementation time for x. More generally, that function would be summing implementation time over all incoming positive influence relations.

It is up to you to decide if such a function is useful in a language. The point is simply that you can define new functions for such purposes. They can aggregate already assigned values into values of a new Value Type. Again, I leave it to the reader as an exercise, to define a language which uses v.ImplTime.

Return now to v.ProgrStatus, where the step called *EstimateDone* was completed for a Fragment x if, in the terminology of this section, there is a value of v.ImplTime assigned to x.

Now, suppose that the team has the rule that, if a Fragment obtains an v.ImplTime value equal or greater than 20 man-hours, then it has to be approved again by

the system designer. Nothing else should change in their work process. Once x is approved, it will immediately enter the product roadmap, because it has the implementation time estimate.

To add this to a language, define a function which is applied for every Fragment that has a v.ImplTime value of 20 or more, and which simply removes the value of v.ProgressStatus of that Fragment, thereby requiring again the approval of the system designer (the language has to have f.chk.progrstatus). The definition of the function is as follows.

Function

Recheck 20 or more
(f.chk.20more)

Input

Model M.

Do

For every Fragment x in M, if x is such that its v.ImplTime is 20 man-hours or more, and its v.ProgrStatus is *EstimateDone*, and since it was added to the model, it was only once been assigned the value *DesignApproved*, then remove the v.ProgrStatus value from x.

Output

A new model M', where all Fragments which were assigned v.ImplTime of 20 man-hours or more, and which were not assigned twice the v.ProgrStatus value *DesignApproved*, now have no v.ProgrStatus value.

Language Services

- **s.WhAppAgain**: Which Fragments in a model M, among all those that have v.ImplTime of 20 man-hours or more, need to be approved again by the system designer? : All Fragments in M, which in M had, and in M' do not have a v.ProgressStatus value.

9.6 Summary on Valuation

Valuation consists of assigning variables to Fragments and relation instances, defining functions over these variables, and given an assignment of values to some of the variables, using the functions to compute values of others.

The section on gave various illustrations of Value Types and how to assign values to parts of models. The key ideas were that to do valuation, it is necessary to choose one or more Value Types for a language. Value Types can be primitive, when they are not defined from other Value Types. v.Satisfaction and v.Approval were primary, for example. There can be derived Value Types, whose values are combinations of values of other Value Types. One of the examples was v.ToDo.

Variables take values from Value Types, and these variables are associated to Fragments and relation instances. One compelling reason for allowing relation instances to be associated with variables, and receive values, is that one can define rules for computing the value on a Fragment. This was illustrated with f.sat and f.sat.x.

Many other topics on valuation are important, and I discuss some of them in subsequent sections, while others remain outside the scope of the tutorial:

- What if random variables need to be assigned to model parts, to say, for example, that there is a probability for a Fragment to get some value? I discuss this in Section 10.
- How to say in models that some values are more or equally desirable than others, on the same Fragment or relation instance, or on other Fragments and relation instances? This is the topic of Section 11.
- How are Value Types and valuation related to truth values in classical and non-classical logics? I will revisit this briefly, for classical logic, in Section 12.

10 Uncertainty and Probability

Overview and Motivation

What if you need models to say that a value assignment is uncertain, and to quantify that uncertainty? What if models need to include random variables?

This section focuses on how to represent that value assignments to model parts are uncertain. This is done by allowing random variables to be associated to model parts, and defining probability spaces for these random variables, so that you can give a probability that the random variable takes a specific value, or any value in a range. The section is organised around the following questions:

- How to represent independent random variables in a model? (Section 10.1),
- What to do when there are dependent random variables in a model? (Section 10.2).

In Section 9.5, the implicit assumption was that there is no uncertainty in value assignments of v.ImplTime. This may be unrealistic. There can be changes in requirements, domain knowledge, and, or specifications, errors in the implementation, or other issues. Stakeholders may be unsure about their estimates.

It was not possible to represent uncertainty about estimates with languages discussed so far. That is, all value assignments were certain. To have more realistic models, a language would need to deliver the following Language Service.

Language Service

UncImplTime: How uncertain is the assignment of the v.ImplTime value to the Fragment x in M?

If a language can deliver s.UncImplTime, then its models can also answer such questions as, for example, "How uncertain (or certain) is it that the implementation time of x will be v?", where v is the v.ImplTime value assigned to x.

The same assumption was implicit when v.Satisfaction values were assigned. If a model assigns the satisfaction value 1 to a requirement such as, say, AddRepEm, then the model says that *all* emergency calls are responded to. The requirement is idealistic, as it is inevitable that, among tens of thousands of calls, some will not be responded to, or not within some prescribed time. But again, there was no way to take a more realistic position, that there is uncertainty about satisfaction values. To avoid this assumption, and allow the representation of more realistic requirements, the language would need to deliver the Language Service below.

Language Service

UncSat: How uncertain is the assignment of the v.Satisfaction value to a Fragment x in M?

v.Approval value assignments can be uncertain as well. A stakeholder may change her mind, and change previously assigned approval values. A model may capture this by describing the uncertainty of an approval value on a Fragment, that is, would deliver the following Language Service.

Language Service

UncApp: How uncertain is the assignment of the v.Approval value to a Fragment x in M?

If a model can answer the above, then it can also answer questions such as, for example, "How certain is it that the approval value of x will change?". This is relevant if you need to decide whether to ask stakeholders for approving again a model, or if the already assigned approval values are stable enough to avoid another round of approval.

To deliver the Language Services above, a language needs to have means for qualifying or quantifying uncertainty. Qualifying amounts to having a scale of qualitative values for describing uncertainty, such as, for example, a scale with only the values "low", "medium", and "high". Quantifying usually means assigning and calculating probability values to events. A language can also combine both, by, for example, having rules that map ranges of probability values to values on a qualitative scale

(say, if probability that a stakeholder changes her approval value on x is at most 0.1, then this corresponds to the value "low" on the qualitative scale), but the challenge in having both is being clear on what they are used for.

10.1 How to Have Independent Random Variables in Models?

To quantify the uncertainty of value assignments, it is necessary to define the probability space of a random variable.

Recall that a *probability space* is a triple $(\mathscr{S}, \mathscr{E}, P)$, where $\mathscr{S}$ is the *sample space*, which includes all possible outcomes of a phenomenon, $\mathscr{E}$ is the set of all *events*, where an event can contain zero or more outcomes, and P is a probability measure, a function which given an event, returns a real value in the range $[0,1]$. If $e \in \mathscr{E}$, then $P(e)$ is called the *probability of* e. If, for example, the phenomenon of interest is the tossing of a perfect coin, then the sample space is $\mathscr{S} = \{H, T\}$, with two outcomes, called H when the "heads" side of the coin is up, and T when the "tails" side is. $\mathscr{E}$ includes all possible combinations of outcomes, that is, it is the power set of $\mathscr{S}$, and the probabilities of events are as follows: $P(\emptyset) = 0$, $P(\{H\}) = 0.5$, $P(\{T\}) = 0.5$, $P(\{H, T\}) = 1$. The probability space would be different if, for example, I was tossing a pair of coins.

An important consequence of allowing random variables in models, is that you have to define a probability space for each variable. And there can be many such variables. For example, suppose that you have a model in L.Rigel, and that all assignments of v.Satisfaction values are uncertain. You know from L.Rigel that, because it has f.sat.inf.pos, f.sat.inf.neg, f.sat, and f.sat.leaf, that you have to assign all satisfaction values to leaf Fragments, and then propagate these values to influence relation instances and Fragments. Now, to quantify the uncertainty of all these value assignments of satisfaction values, observe that you have as many random variables, as there are assignments of satisfaction values. This is because if x is a Fragment or relation instance, then there is a random variable x.v.Satisfaction, and you need a probability space for it. So if $\langle x, \text{v.Satisfaction}, 1\rangle$, or equivalently, x.v.Satisfaction = 1, then you need a probability space for x.v.Satisfaction in order to compute the probability of it getting a specific satisfaction value. If that value is 1, you need its probability space if you want a value for $P(x.\text{v.Satisfaction} = 1)$, which is, given my notational conventions in this tutorial, the same as wanting the value of $P(\langle x, \text{v.Satisfaction}, 1\rangle)$.

Exercise 26: Define a function which returns the probability of a value assignment

Take any language defined so far, and which can represent the assignment of a v.Satisfaction value to Fragments and relation instances. How would you enable that language to represent the probability of a v.Satisfaction value assignment?

To deliver s.UncImplTime, a language needs to associate a random variable x.v.ImplTime to every Fragment x. In addition, each random variable will come with its own probability space, which includes the function that returns the probability of a specific value of x.v.ImplTime.

Recall that v.ImplTime is a positive real. For any Fragment x, then, and in the terminology of probability spaces, x.v.ImplTime takes a value from the sample space $[0,\infty)$, and any such value is an outcome. Any event of interest is any one of these outcomes. Furthermore, as it takes a real value, x.v.ImplTime has a continuous probability distribution, and has to have a probability density function, which is denoted *pdf*(x.v.ImplTime) below.

For example, perhaps v_x follows a normal (Gaussian) distribution with a mean of 10 man-hours, and a standard deviation of 2 man-hours, so that $pdf(v_x) = (1/2\sqrt{2\pi})e^{-(v_x-10)^2/8}$. But, there can be another Fragment y, which has v_y as its random variable, and v_y may have a completely different probability density function (not the one for normal distribution).

Regardless of the specifics of the probability density function, the uncertainty of a value assigned to x.v.ImplTime is quantified with a probability measure, whereby the probability that implementation time x.v.ImplTime is in the interval $[a, b]$ is given by

$$P[a \le v_x \le b] = \int_a^b pdf(v_x)\, dv.$$

Similar stories can be told for v.Satisfaction and v.Approval. If you want to indicate in a model that you are unsure about the satisfaction or approval value on a Fragment or relation instance x, then associate the random variable to x, and define the probability space for it.

How does the discussion influence the Language Modules that you define in a language. It is important to see that there are two ways to use random variables.

1. *Probability measurement* consists of doing the following. Start by assigning values to Fragments, and then calculate the probability of these values. For example, if the estimate of implementation time for a Fragment x is 13 man-hours, then calculate the probability $P[x.\text{v.ImplTime} \le 13]$. The probability value thus quantifies the uncertainty of this estimate, with the slight adjustment that it gives the probability that implementation time for x will be at most 13 man-hours, and not exactly 13 man-hours. The adjustment is due to *pdf*(x.v.ImplTime) being a continuous function over reals, so that $P[x.\text{v.ImplTime} = c] = 0$, for any constant c. If x.v.ImplTime is discrete, and has a probability mass function instead of *pdf*(x.v.ImplTime), then it makes sense to compute $P(x.\text{v.ImplTime} = 13)$.

2. *Simulation:* Do not assert the value of a random variable x.v.ImplTime, but generate a value for it by simulation. So instead of assigning yourself, or asking someone for a value of implementation time, obtain that value through simulation which generates random values that satisfy the specifics of the probability density function, or probability mass function of the random variable.

Both approaches add new functions and Value Types to a language. The measurement approach adds functions which return probability values, while the simulation approach adds functions which return a value of a random variable, produced by simulation. If a model has n random variables, then there have to be n probability spaces, one per random variable. I introduce the convention that each probability space defines a new Value Type, which is named as follows: if x.v.ImplTime is the random variable, then there has to be the Value Type v.prob(x.v.ImplTime) defined by the probability space for x.v.ImplTime. Illustrations are below.

For the measurement approach, a language can have a generic function which takes a probability space and returns a probability value, and is defined as follows.

Function

Assign probability value
(f.prob.asg)

Input

- Assignment either of a single value $\langle x, \text{v.T}, w\rangle$ or of a range $\langle x, \text{v.T}, w_1 \leq v \leq w_2\rangle$ to random variable of Value Type T on Fragment x, and
- Value Type v.prob(x.v.T), defined by the probability space $(\mathscr{S}, \mathscr{E}, P)$ for the random variable x.v.T.

Do

If the input is $\langle x, \text{v.T}, w\rangle$, then $p = P(x.\text{v.T} = w)$. If input is $\langle x, \text{v.T}, w_1 \leq v \leq w_2\rangle$, then $p = P[w_1 \leq v_x \leq w_2]$.

Output

$\langle x, \text{v.prob}(x.\text{v.T}), p\rangle$, that is, the assignment of a probability value, which is the probability that $\langle x, \text{v.T}, w\rangle$ or $\langle x, \text{v.T}, w_1 \leq v \leq w_2\rangle$, depending on the input to f.prob.asg.

Language Services

- **s.WhProbability**: If the probability space for the random variable x.v.T is $(\mathscr{S}, \mathscr{E}, P)$, then what is the probability that x.v.T $= w$ if w is given, or that x.v.T $\in [w_1, w_2]$, if $[w_1, w_2]$ is given? The value assignment $\langle x, \text{v.prob}(x.\text{v.T}), p\rangle$ returned by f.prob.asg.

As it is defined above, f.prob.asg is not specific to particular Value Types, or to discrete or continuous random variables. The function assumes that a probability space is already defined for a random variable x.v.T, and f.prob.asg returns, using the probability function defined for that space, the probability value. f.prob.asg is defined rather loosely, since it says nothing about, for example, how it is ensured that the input value or range for x.v.T matches the properties of the probability space, that is, makes sense for the given probability space (for example, if a single value is input to f.prob.asg, then f.prob.asg will return a zero value if the random variable is not discrete).

If the language does allow the definition of random variables, a major difficulty is to design relevant probability spaces, be it because of biases [127], or because the required data is hard to find. For instance, it may not be clear at all where to look for useful data, in order to define the probability space for implementation time of some Fragment x.

The simulation approach also involves adding one or more functions to a language. For example, let the aim be to generate values for random variables that follow the normal distribution. A function is needed, which takes the mean and standard deviation parameters of the normal distribution that the variable follows. The function may apply, for example, the Box-Muller method [19] to generate and output a value for the random variable.

A model can include random variables, such as some x.v.Tq, whose probability is determined by a joint probability distribution of two or more other random variables, say x_1.v.T1,…, x_n.v.Tm in the same model.

For illustration, suppose that the probability space v.prob(x.v.Satisfaction) is such that the probability of x.v.Satisfaction is given by the joint probability distribution of the variables

$$x_1.\text{v.Satisfaction}, \ldots, x_n.\text{v.Satisfaction}.$$

If they are all independent variables, then

$$P(x.\text{v.Satisfaction} = b) = P(x_1.\text{v.Satisfaction} = a_1) \cdot \ldots \cdot P(x_n.\text{v.Satisfaction} = a_n).$$

If the model says that every Fragment $x_1, \ldots x_n$ has to be satisfied, in order for x to be satisfied, then

$$P(x.\text{v.Satisfaction} = 1) = P(x_1.\text{v.Satisfaction} = 1) \cdot \ldots \cdot P(x_n.\text{v.Satisfaction} = 1).$$

The above can be shown as a graph, by having an edge from each of the random variables x_i.v.Satisfaction to x.v.Satisfaction. Another approach is to reuse instances of another relation, some r.K, which already generates a graph. This consists of assuming that each r.K instance also indicates that the probability of some value assignment to a Fragment is the product of the probabilities of specific value assignments on other Fragments. The following example illustrates this.

Example 10.1. Recall that a language can have influence relations to show how a satisfaction value of a Fragment or relation instance influences that of another. These relations can be used to define the joint probability distribution, to use to compute the probability of satisfying a Fragment. Namely, a language can have a rule which says that, if to satisfy x, it is necessary to satisfy all Fragments, say $x_1, \ldots, x_n$ connected via r.inf.pos to x, then the probability of satisfying x is given by the joint probability distribution of the random variables of v.Satisfaction, assigned to $x_1, \ldots, x_n$. The rule can be added to a language with f.prob.prod below.

Function

Compute probability that a Fragment is satisfied using probabilities that Fragments which influence it positively are (f.prob.sat.ind)

Input

Fragment x and model M.

Do

1. Let $\{y_1, \ldots, y_n\} \subseteq$ r.inf.pos be all r.inf.pos to x in M.
2. Let

$$\langle y_1, \text{v.prob}(y_1.\text{v.Satisfaction}), P(y_1.\text{v.Satisfaction} = 1)\rangle,$$
$$\ldots,$$
$$\langle y_n, \text{v.prob}(y_n.\text{v.Satisfaction}), P(y_n.\text{v.Satisfaction} = 1)\rangle$$

be probability values that each y_1 will take the v.Satisfaction value 1.

3. If y_1.v.Satisfaction, $\ldots, y_n$.v.Satisfaction are independent random variables, and the probability of x.v.Satisfaction is given by the joint probability distribution of y_1.v.Satisfaction, $\ldots, y_n$.v.Satisfaction, then

$$P(x.\text{v.Satisfaction} = 1) = \prod_{i=1}^{n} P(y_i.\text{v.Satisfaction} = 1).$$

Output

$\langle x, \text{v.prob}(x.\text{v.Satisfaction}), P(x.\text{v.Satisfaction} = 1)\rangle$.

Language Services

- **s.WhProbSatInd**: What is the probability of satisfying x, if $y_1, \ldots, y_n$ all positively influence x in M, the probability of satisfying each of y_1 is independent from the probability of satisfying any other y_j, and the probability of satisfying x is given by the joint probability distribution function of satisfying all Fragments which influence x? : $\langle x, \text{v.prob}(x.\text{v.Satisfaction}), P(x.\text{v.Satisfaction} = 1)\rangle$.

The language below allows random variables in models, and has f.prob.asg and f.prob.sat.ind.

Language

Adhara

Language Modules

F, **T**, **V**, r.inf.pos, r.inf.neg, f.brel2g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat, f.sat.leaf, f.prob.asg, f.prob.sat.ind

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, F = c.r∪c.k∪c.s and c.r∩c.k∩c.s = ∅. Influences are over Fragments, r.inf.pos ⊆ F × F, r.inf.neg ⊆ F × F. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\begin{aligned}\mathbf{V} \quad \subseteq \quad & W \times \{\text{v.Satisfaction}\} \times \text{v.Satisfaction} \\ & \cup\, W \times \{\text{v.prob}(x.\text{v.Satisfaction}) \mid xW\} \times [0,1], \\ & \text{where } W = \text{F} \cup \text{r.inf.pos} \cup \text{r.inf.neg}.\end{aligned}$$

The above says that any value assignment is the assignment of a v.Satisfaction to a Fragment or influence relation instance, or the assignment of a value from a range $[0,1]$ of reals, to x.v.prob(x.v.Satisfaction), where, again, x is a Fragment or an influence relation instance. So the first part of **V** are assignments of satisfaction values, and the second part are assignments of the probability of satisfaction value assignments.

The language has many Value Types,

$$\begin{aligned}\mathbf{T} \quad = \quad & \{\text{v.Satisfaction}\} \cup \{\text{v.prob}(x.\text{v.Satisfaction}) \mid x \in W\}, \\ & \text{where } W = \text{F} \cup \text{r.inf.pos} \cup \text{r.inf.neg}.\end{aligned}$$

with v.Satisfaction = $\{1,0\}$ and v.prob(w.v.Satisfaction) = $[0,1]$, for every $w \in W$.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{aligned}\alpha \quad &::= \quad x \mid y \mid z \mid \ldots \\ \beta \quad &::= \quad r \mid k \mid s \\ \gamma \quad &::= \quad \beta(\alpha) \\ \delta \quad &::= \quad (\gamma, \gamma) \\ \epsilon \quad &::= \quad \langle \alpha, \zeta, \eta \rangle \\ \phi \quad &::= \quad \gamma \mid \delta \mid \epsilon\end{aligned}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in$ F. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. δ symbols denote positive and negative influence relations. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in$ **T**. η denotes a value of a Value Type, and as there is one Value Type, then $\mathscr{D}(\eta) \in$ v.Satisfaction. ϵ symbols denote value assignments, $\mathscr{D}(\epsilon) \in$ **V**.

Language Services

Those of relations and functions in the language, and s.SatReq.

Figure 18 shows a model in L.Adhara, when all the random variables of type v.Satisfaction are independent. Each of these variable is denoted $v[m]$, where m is the Fragment identifier. Each random variable is of type v.Satisfaction. There is the assignment of a probability value to each Fragment. Each indicates the probability that the Fragment is satisfied, that the value of the variable is 1. The probability that AmbArrIncLoc is satisfied is equal to the joint probability of satisfying all other Fragments shown in the Figure. •

10.2 What If Random Variables Are Dependent?

This section drops two assumptions that were made in Section 10.1: (i) that events are independent, so that the occurrence of one does not influence the probability of another to occur, and (ii) that random variables are independent, or in other words, that the occurrence of events of one of the variables does not influence the probability of the events of the other random variable.

Exercise 27: Define a language which can represent dependent random variables

How would you change L.Adhara if its models need to represent dependent random variables? How would you compute the probability of an event in a model of that new language, if that probability depends on the occurrence of other events that can be represented in the model?

A language can use Bayesian networks [101, 22] to represent dependency between

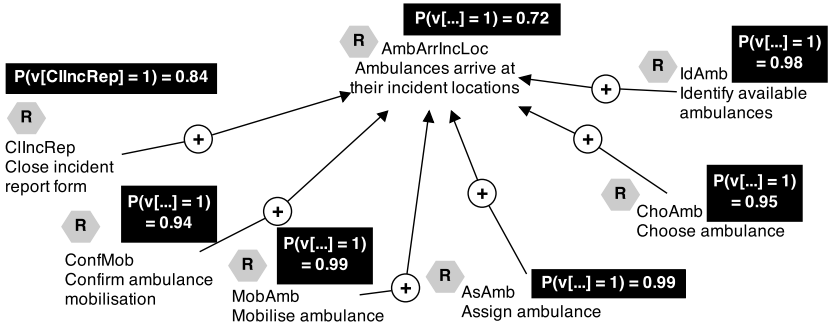


Figure 18: A model in L.Adhara with assignments of probability values.

random variables, and to compute probabilities of their events.

A Bayesian network is a directed acyclic graph (V, E), where V is a set of random variables and E of edges. There is an edge from $v_1 \in V$ to $v_2 \in V$, iff $P(v_1) \neq P(v_1 \mid v_2)$, that is, the probability of an event of v_1 is different from the probability of the event, given the occurrence of an event of v_2. If there are two edges to v_1, for example (v_3, v_1) and (v_2, v_1), then this says that $P(v_1) \neq P(v_1 \mid v_2, v_3)$ and that $P(v_1 \mid v_2) \neq P(v_1 \mid v_2, v_3)$. More generally, in a Bayesian network, every random variable is dependent only on its direct parent variables. In an edge (v_2, v_1), v_2 is a direct parent of v_1, while if there another edge (v_3, v_2), then v_3 is an indirect parent of v_1, and so, $P(v_1 \mid v_2) = P(v_1 \mid v_2, v_3)$.

An important property of Bayesian networks is that, to give the joint probability distribution for all random variables in the network (that is, to have the probability value for all events, of all random variables in the network), it is enough to specify only the probability values for all events of all root random variables (those with no parents), and the conditional probability values for all events of all non-root random variables, for all possible combinations of events of their direct parents. While this can require considerable work as well, it is less than the $2^{|V|-1}$ probability values, which would otherwise need to be defined.

There are at least two approaches to enabling a language to represent Bayesian networks in its models, provided that this language does allow associating random variables to Fragments. The approach in Section 10.2.1 ignores relations which may exist in the language. So there is no mapping between a Bayesian network and some graph that a relation gives. This also means that there are no existing graphs in a model in that language, which can be used to produce the corresponding Bayesian network automatically. The approach in Section 10.2.2 automatically generates a Bayesian network, based on a graph in a model of the language. I will consider both options below.

10.2.1 If Bayesian Networks Ignore Existing Relations in a Language

The first approach consists of adding a function which takes all random variables assigned to Fragments in a model, and produces a Bayesian network over these variables (note that the network does not need to be a connected graph). The function is defined as follows.

Function

Make a Bayesian Network
(f.make.baynet)

Input

Set X of Fragments.

Do

Let:

- V_X be the set of all random variables, at most one per Fragment in X,
- (V_X, E) be a Bayesian network with no edges,

Then:

1. for every pair x, y in V_X, if $P(x) \neq P(x \mid y)$, then add an edge to E, directed from y to x,
2. for every random variable which is a root node in (V_X, E), define probability values for all its possible events,
3. for every random variable which is not a root node in (V_X, E), define conditional probability values for all events of all non-root random variables, for all possible combinations of events of their direct parents.

Output

Bayesian network (V_X, E).

Language Services

- **s.WhProbBN**: What is the probability of an event e of variable v_x to occur, according to the Bayesian network (V_X, E)? : $P(v_x = e)$ obtained by evaluating the Bayesian network (V_X, E).

10.2.2 If Bayesian Networks are Derived from Existing Relations in a Language

Given a model in a language, f.make.baynet only uses the random variables assigned to Fragments in that model. It ignores all else that may be said in the model, such as the relations that the Fragments are in.

When the aim is to reuse more of the information in a model, then it may be relevant to derive (part of) a Bayesian network from some relation in a language.

For illustration, recall that influence relations exist when the satisfaction value of a Fragment depends on satisfaction values of others. If I decide that positive influence relations should be interpreted as giving probability dependence between random variables assigned to Fragments in these relations, then I can map a graph over influence relation instances to a Bayesian network. The idea is that if there is a positive influence from Fragment y to x, and v_y and v_x are the random variables associated to, respectively y and x, then there is an edge in the Bayesian network where v_x and v_y are nodes. The following function does this.

Function

Make a Bayesian Network from r.inf.pos instances
(f.map.inf.pos.baynet)

Input

G(X, r.inf.pos), where X is a set of Fragments.

Do

Let:

- V_X be the set of all random variables, at most one per Fragment in X,
- (V_X, E) be a Bayesian network with no edges.

Then:

1. for every edge (y, x) in G_{I+}, add an edge (v_y, v_x) to E, where v_x and v_y are random variables assigned to, respectively, x and y,
2. for every random variable which is a root node in (V_X, E), define probability values for all its possible events,
3. for every random variable which is not a root node in (V_X, E), define conditional probability values for all events of all non-root random variables, for all possible combinations of events of their direct parents.

Output

Bayesian network (V_X, E).

Language Services

s.WhProbBN.

Example 10.2. Figure 19 gives a simple and hypothetical example of applying f.map.inf.pos.baynet to a graph G(X, r.inf.pos).

Every Fragment in the graph G(X, r.inf.pos) on the left-hand side of the Figure has an associated random variable of the format $v[\ldots]$. This is shown with dashed lines. The right-hand side of the Figure shows a Bayesian network, where edges are marked "B", made by applying f.map.inf.pos.baynet to the graph G(X, r.inf.pos) on the left-hand side of the figure. Hypothetical probability values to root nodes, and the conditional probability values to the one non-root node were assigned manually. •

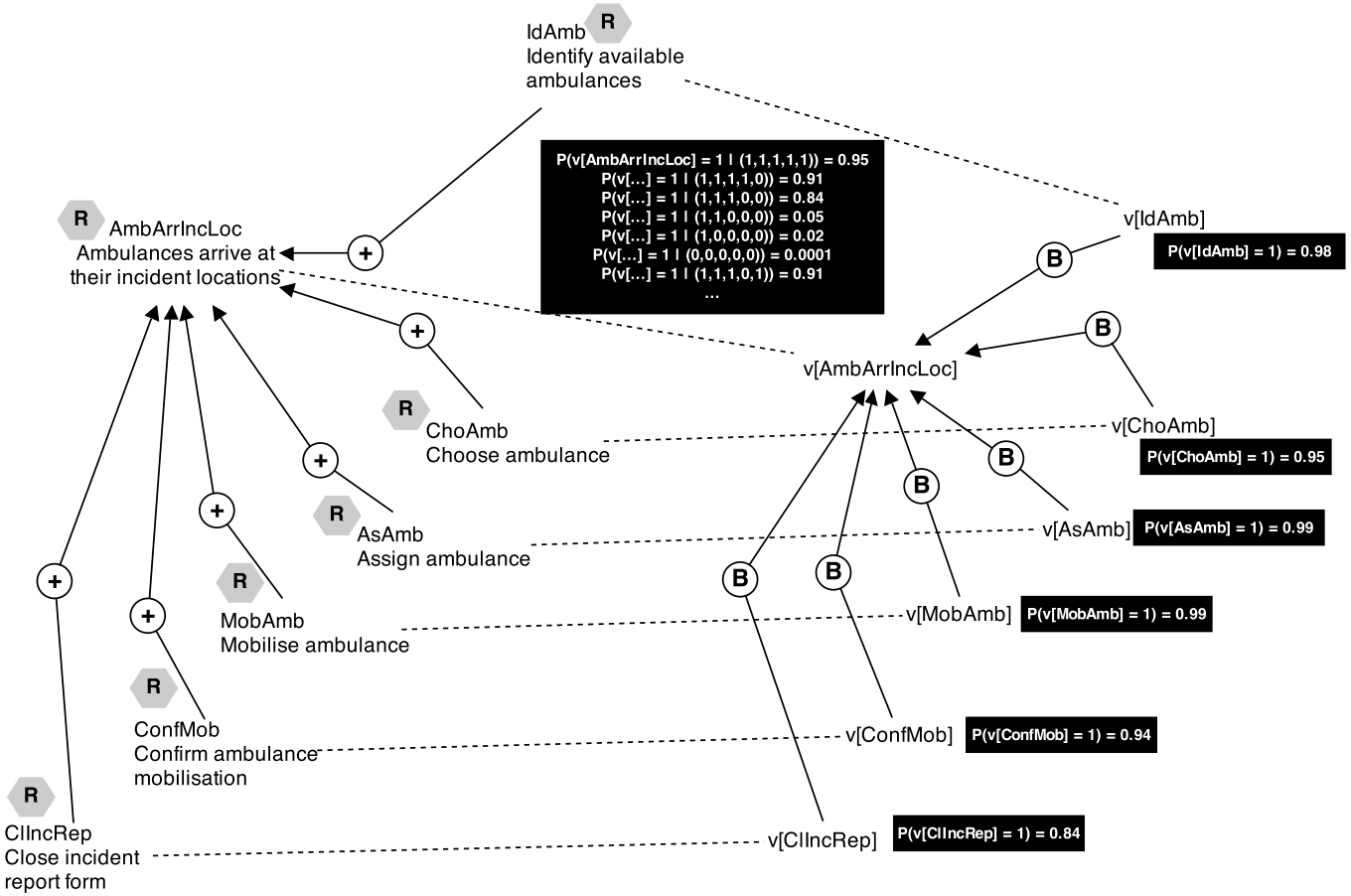


Figure 19: Bayesian network from positive influence relation instances.

11 Preferences

Overview and Motivation

Given a set of Outcomes, how do you find the "best" one? What tells you, in a model, if an Outcome is "better" than another?

It may be more desirable to stakeholders that incident reports are managed via the dispatching software, than having it done via other software. Each of these, in turn, may be more desirable than to fill out and keep paper incident reports. Some Outcomes will assign v.Satisfaction value 1 to using dispatching software to manage incident reports, and will, with regards to how incident reports are managed, be more desirable to other Outcomes in which assign 0 to this Fragment.

Choosing the "best" Outcome can be done by indicating the relative desirability of value assignments, that is, *preferences*. Preferences can be associated to different criteria, such as cost, time to implement, ease of use, and so on. Given preferences and the criteria in a model, the aim is to somehow use them to order Outcomes. This involves various activities, such as eliciting preferences, finding criteria, inferring missing preferences and orders over Outcomes.

This section focuses on how to enable languages to represent preferences and criteria, and then identify the best Outcome. I discuss the following questions.

1. What are preferences and criteria? (Section 11.1),
2. Why and how to use preferences over values of a single Value Type, on a single Fragment or relation instance? (Section 11.2),
3. Why and how to use preferences over values of a single Value Type, on various Fragments or relation instances? (Section 11.3),
4. Why and how to use preferences over values of a different Value Types, on a single Fragment or relation instance? (Section 11.4),
5. Why and how to use preferences over values of a different Value Types, on various Fragments or relation instances? (Section 11.5),
6. Where to find Criteria in requirements? (Section 11.6),
7. How to use preferences to find best Outcomes in models? (Section 11.7).

You rarely have a total order over Outcomes. There may be so many value assignments, so it is not feasible to elicit all the comparisons needed to define the total order. It can also happen that you have no one to elicit them from. Or you may, but perhaps you do not trust that these comparisons will remain unchanged. Stakeholders need not know the values or Outcome to prefer, especially if it is unclear how these values and Outcomes translate to their specific context.

To define the total order, you can elicit pairwise comparisons of some value assignments, and, or Outcomes, and somehow deduce the remaining comparisons that you need to define the total order. In the worst case, you would need to elicit all possible comparisons among pairs of value assignments. Such comparisons are called *preferences*, each saying that some value assignment v_1 is more desirable than some other v_2.

Preferences are associated to Criteria, such as "low cost", "short implementation time", "positive effect on the scalability of the system", and so on. For example, it may be more desirable that the average time to respond to incidents is 12 minutes than 16 minutes, and the criterion in this case may be called "lower average time to respond to an incident". However, it may be that achieving an average of 12 minutes is more costly (requires more ambulances, more personnel, and so on) than achieving an average of 16 minutes. The two value assignments are thus compared over two criteria, one being the average time to respond to an incident, the other the cost of the future system.

In short, the idea is that you would discover and elicit preferences incrementally and often partially. You may decide to stop, when you have enough of them to approximate the total order over Outcomes, and thereby highlight the best one.

This absence of information about preferences, and its incremental discovery and elicitation, is also a major reason to make languages which can represent preferences. As you elicit new preferences, you add them to a model, and you can analyse how they relate to already existing preferences, over the same criteria, or if you need to add new criteria as well. You can evaluate if a given model includes enough information on preferences and criteria, to produce a partial or total order, over many criteria, over the Outcomes.

11.1 What Are Preferences and Criteria?

There is considerable research on preferences in philosophy [108, 61], economics [80, 126, 128, 135, 20, 122, 97, 92], operations research [47, 55, 45], and artificial intelligence [5, 41, 39]. In Section 11.1.1, I recall common ideas about two core preference relations, called strict preference and indifference. In Sections 11.1.2–11.1.5, I introduce the preference-related terminology specific to this tutorial, and relate it to the core preference relations.

11.1.1 Core Preference Relations

If you ask which of A and B is more desirable, you can expect any one of three answers. A, for example, may be more desirable than B, that is, better than B, or *vice versa*. In that case, there is the so-called *strict preference* for one over the other. Another answer is that A and B are equally desirable, neither is better than the other. This is a case of being *indifferent* to A and B. Finally, A and B can be incomparable in terms of desirability, in which case there is no preference between them.

Strict preference and indifference are two core preference relations [62], and any other is a derived preference relation. When I write "core preference relations", I am referring to strict preference and indifference relations. When I want to be specific, I will write "strict preference" or "indifference".

Strict preference is usually an irreflexive, antisymmetric, and transitive binary relation. Indifference is reflexive, symmetric, and transitive.

Core preference relations can, but need not be *complete* over a domain. A strict preference relation is complete for its domain iff there is an instance thereof between every pair of elements in that domain. This is different than the usual approach, in that a preference relation can be complete if there is either strict preference *or indifference* between any pair of elements in the domain. I do this in order to simplify the discussion in this tutorial.

Completeness is a desirable property when you want to establish a total order over Alternatives or Combinations. But as I said earlier, it can be difficult to find enough information to achieve it. There is considerable work on the elicitation of preferences [23], which I leave to you to explore.

All things in the domain of a preference relation are assumed to be *comparable*. This means that there are strict preference, or indifference, or both relation instances between any two pairs of things in the domain.

In addition to the above, it is also usually assumed that all things in the domain of a preference relation are mutually exclusive. That is, none is part of another, and none is compatible with another.

11.1.2 Domains of Preference Relations

In this tutorial, preference relations are over value assignments. The domain of a preference relation includes only value assignments.

Fragments (and the same applies to relation instances), when taken independently of values, are not members of domains of preference relations. If I write that "Fragment x is strictly preferred to a Fragment y" then it is not clear if I am trying to say that "*satisfying* Fragment x is strictly preferred to a *satisfying* Fragment y", or that "*including in the model the* Fragment x is strictly preferred to a *including in the model the* Fragment y", or both, or none of these, but something else. Having only value assignments in preference domains allows me to be more precise, without losing the ability to say either of these. The statement "*satisfying* Fragment x is strictly preferred to a *satisfying* Fragment y" is a preference over satisfaction values, while "*including in the model the* Fragment x is strictly preferred to a *including in the model the* Fragment y" can be a preference over acceptability value assignments.

11.1.3 Preference Relation Instances and Domains

An instance of some preference relation (core or not), called Pref, for example, is written

$$(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle) \in \mathsf{r.Pref}$$

to say that the value assignment $\langle x_i, t_j, v_k\rangle$ is strictly preferred to $\langle x_l, t_p, v_q\rangle$ according to the preference relation r.Pref.

x_i can, but need not be equal to x_l, and the same goes for t_j and t_p. That is, there can be preference relations which are not over values of the same Value Type, or over values assigned to the same Fragment or relation instance. In order to write about domains which include value assignments with different Value Types, and over different Fragments and relation instances, I will use the following notational conventions.

- $x^{(n)}$ denotes the set $\{x_1, \ldots, x_n\}$.
- $\langle x^{(n)}, t_j, v_k\rangle$ denotes a set of n assignments of the same value v_k, of Value Type t_j, to every member of $x^{(n)}$. So if I write $\langle x^{(n)}, \mathsf{v.ImplTime}, 5\text{man-days}\rangle$, then I am saying that each of $x_1, \ldots, x_n$ takes 5 man-days to implement.
- $\langle x^{(n)}, t_j, v^{(m)}\rangle$ is a set of value assignments, where each member of $x^{(n)}$ gets one or more values from $v^{(m)}$, whereby all values in $v^{(m)}$ are of the same Value Type t_j.
- $\langle x^{(n)}, t^{(u)}, v^{(m)}\rangle$ is the set of value assignments, where each member of $x^{(n)}$ gets one or more values from $v^{(m)}$, and each of these values is of one of the Value Types in $t^{(u)}$.

I use the above in Section 11.1.4 to define different types of preference relations. For example, there will be one preference relation type, whose domain is the set $\langle x^{(n)}, t_j, v_k\rangle$, and there will be another type, whose domain is $\langle x^{(n)}, t^{(u)}, v^{(m)}\rangle$.

11.1.4 Some Interesting Preference Relations

Imposing different constraints on the content of the domain of preference relations gives different categories of such relations. Each category gives different answers to the following questions.

- Is the Fragment or relation instance the same in all value assignments in the domain?
- Is the Value Type the same in all value assignments in the domain?
- Can the domain include value assignments to both Fragments and relation instances?

Table 2 shows some possible combinations of answers to these questions. In the Table, x is a Fragment, and r a relation instance. Each row is a single combination of answers, each column one preference relation category. The symbol "•" indicates all answer combinations that a preference relation allows. Note that some preference relations in that Table allow several combinations of answers.

For example, the domain of the preference relation P_A includes assignments of different values, of the same Value Type, to the same Fragment. In contrast, the domain of P_J includes assignments of different values, of the same Value Type, to different Fragments and/or relation instances (and the domain can include both Fragments and relation instances).

In Table 2, the domains of some preference relations are subdomains of others. For example, all preference relation instances that can be written with P_A or P_C can also be written with P_G. Also, note the domains of P_L and P_M are equivalent, so P_M can be removed from the Table.

In the rest of this section, I concentrate only on the preference relation categories P_I, P_J, P_K, and P_L. I use the following names for them. P_I relations are called Local Preferences (Section 11.2), P_J are Bridge Preferences (Section 11.4), P_K are Mixed Local Preferences (Section 11.3), and P_L are Mixed Bridge Preferences (Section 11.5).

11.1.5 Criteria

A preference relation is, in this tutorial, always associated to a Criterion. A Criterion C is a function over value assignments, such that if there is a preference relation instance $(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle)$, and it is associated to the Criterion C, then

$$C(\langle x_i, t_j, v_k\rangle) > C(\langle x_l, t_p, v_q\rangle)$$

that is, the value of $C(\langle x_i, t_j, v_k\rangle)$ is greater than that of $C(\langle x_l, t_p, v_q\rangle)$. A Criterion is, then, a function which returns a greater value for more desirable Alternatives.

Every Criterion can have its own Value Type, which may, but need not be related in some way to the Value Types. Above, suppose that $\langle x_i, t_j, v_k\rangle$ is a cost value, and $\langle x_l, t_p, v_q\rangle$ an estimate of implementation time. The preference $(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle)$ thus says that observing a specific cost is strictly more preferred than to observe a specific implementation time.

Criteria specialise preference relations, in that there can be a preference relation specific to cost, another one specific to implementation time, and so on. I will write

$$(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle) \in \mathsf{r.Pref.C}$$

if $(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle)$ is an instance of some preference relation called Pref, associated to the Criterion C.

11.2 Why Local Preferences?

A Local Preference instance is a strict preference over values of a single Value Type, assigned to a single Fragment, or a single relation instance. That is, every Local Preference instance is a pair $(\langle x, t, v_i\rangle, \langle x, t, v_j\rangle)$, where x is a Fragment or a relation instance, and v_1 and v_2 are values of the same Value Type t. The following relation can be used to enable a language to represent Local Preferences.

Relation
Local Preference (r.pref.loc.c)
Domain & Dimension r.pref.loc.c $\subseteq \langle x, t, v^{(n)}\rangle$, where c is a Criterion, and $\langle x, t, v^{(n)}\rangle$ is a set of n value assignments $\langle x, t, v_i\rangle$ on the same Fragment or relation instance x, of values of the same Value Type t.
Properties $(\langle x, t, v_i\rangle, \langle x, t, v_j\rangle \in$ r.pref.loc.c if $\langle x, t, v_i\rangle$ is strictly more desirable to $\langle x, t, v_j\rangle$ on the Criterion c.
Reading $(\langle x, t, v_i\rangle, \langle x, t, v_j\rangle \in$ r.pref.loc.c reads "$\langle x, t, v_i\rangle$ is strictly preferred to $\langle x, t, v_j\rangle$ on the Criterion c".
Language Services • **s.IsLocPref**: Is $\langle x, t, v_i\rangle$ is strictly preferred to $\langle x, t, v_j\rangle$ on the Criterion

Table 2: Some categories of preference relations.

Domain	Preference relations categories P_A	P_B	P_C	P_D	P_E	P_F	P_G	P_H	P_I	P_J	P_K	P_L	P_M
$\langle x, t, v^{(m)} \rangle$	•	-	-	-	-	-	-	-	•	-	-	-	•
$\langle r, t, v^{(m)} \rangle$	-	•	-	-	-	-	-	-	•	-	-	-	•
$\langle x^{(n)}, t, v^{(m)} \rangle$	-	-	•	-	-	-	-	-	-	•	-	-	•
$\langle r^{(n)}, t, v^{(m)} \rangle$	-	-	-	•	-	-	-	-	-	•	-	-	•
$\langle x, t^{(u)}, v^{(m)} \rangle$	-	-	-	-	•	-	-	-	-	-	•	-	•
$\langle r, t^{(u)}, v^{(m)} \rangle$	-	-	-	-	-	•	-	-	-	-	•	-	•
$\langle x^{(n)}, t^{(u)}, v^{(m)} \rangle$	-	-	-	-	-	-	•	-	-	-	-	•	•
$\langle r^{(n)}, t^{(u)}, v^{(m)} \rangle$	-	-	-	-	-	-	-	•	-	-	-	•	•

c? : Yes, if $(\langle x, t, v_i \rangle, \langle x, t, v_j \rangle \in$ r.pref.loc.c.

Local Preferences are the simplest kind of preference relations discussed in this tutorial. Being local to a Fragment or relation instance, and over values of a single Value Type, they do not pose problems of interpretation that, for example, Bridge Preferences will in Section 11.4.

Exercise 28: Define a language which can represent Local Preferences in its models

Define a language which can show assignments of v.Satisfaction values to Fragments and relation instances, and can represent Local Preferences over these value assignments.

The following example illustrates how to enable a language which resembles L.Ankaa to represent Local Preferences.

Example 11.1. Suppose that you want to represent r.pref.loc instances over value assignments in L.Ankaa models, in which all Fragments are categorised as either requirement, domain knowledge, or specification. The following language does this.

Language

Bellatrix

Language Modules

F, **T**, **V**, **C**, r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat, f.sat.leaf, r.pref.loc.c

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, F = c.r∪c.k∪c.s and c.r∩c.k∩c.s = ∅. Influences are over Fragments, r.inf.pos ⊆ F × F, r.inf.neg ⊆ F × F. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\mathbf{V} \subseteq (\mathsf{F} \cup \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg}) \times \mathbf{T} \times \mathsf{v.Satisfaction}$$

where **T** = {v.Satisfaction. For every Criterion, preferences are over value assignments, r.pref.loc.c ⊆ **V** × **V**, for every $c \in \mathbf{C}$.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every

ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & r \mid k \mid s \\ \gamma & ::= & \beta(\alpha) \\ \delta & ::= & (\gamma, \gamma) \\ \epsilon & ::= & \langle \alpha, \zeta, \eta \rangle \\ \theta & ::= & (\epsilon, \epsilon) \\ \phi & ::= & \gamma \mid \delta \mid \epsilon \mid \theta \end{array}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in$ F. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. δ symbols denote positive and negative influence relations. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in$ **T**. η denotes a value of a Value Type, and as there is one Value Type, then $\mathscr{D}(\eta) \in$ v.Satisfaction. ϵ symbols denote value assignments, $\mathscr{D}(\epsilon) \in$ **V**. θ symbols denote preference relations

$$\mathscr{D}(\theta) \in \bigcup_{c \in \mathbf{C}} \text{r.pref.loc.c.}$$

Language Services

Those of relations and functions in the language.

Figure 20 is a visualisation of a model in L.Bellatrix. Instances of r.pref.loc are shown as edges labelled "LP". The model includes the following Local Preferences.

- v.IncRepErr is the percentage of incident reports which include errors. Observing 1% of erroneous incident reports is strictly preferred to 5%, and that 5% is strictly preferred to 7%:

  (⟨FillIncRep, v.IncRepErr, 1%⟩, ⟨FillIncRep, v.IncRepErr, 5%⟩) ∈ r.pref.loc.IncRepErr
  (⟨FillIncRep, v.IncRepErr, 5%⟩, ⟨FillIncRep, v.IncRepErr, 7%⟩) ∈ r.pref.loc.IncRepErr

- v.AddIncTm is the average time to address a reported incident. It is preferred to address a reported incident on average in 12min than in 16min:

  (⟨AddRepEm, v.AddIncTm, 12min⟩,
  ⟨AddRepEm, v.AddIncTm, 16min⟩) ∈ r.pref.loc.AddIncTm

- v.Satisfaction is a binary Value Type, indicating if a Fragment or relation instance is satisfied (value 1) or not. It is strictly preferred that callers do not report incorrect incident locations, as not satisfying IncCalRep is strictly preferred to satisfying it:

  (⟨IncCalRep, v.Satisfaction, 0⟩, ⟨IncCalRep, v.Satisfaction, 1⟩) ∈ r.pref.loc.Satisfaction

- v.CalRepErr is the percentage of calls, in which callers reported incorrect incident location. It is strictly preferred to observe 15% than 21% of such calls:

  (⟨IncCalRep, v.CallRepEr, 15%⟩, ⟨IncCalRep, v.CallRepEr, 21%⟩) ∈ r.pref.loc.CallRepEr

- v.Acceptability is a binary Value Type, with 1 if a relation instance is acceptable, 0 if not. It is preferred that IncCalRep is not acceptable:

  (⟨(IncCalRep, IdIncLoc), v.Acceptability, 0⟩,
  ⟨(IncCalRep, IdIncLoc), v.Acceptability, 1⟩) ∈ r.pref.loc.Acceptability

To clarify terminology, note that there are six instances of r.pref.loc and five Criteria above. This example focuses on Local Preferences, so it is not important which other preference relations are shown in Figure 20. •

11.3 Why Mixed Local Preferences?

Mixed Local Preferences are preferences which can be over values of different Value Types, on the same Fragment or relation instance. Every instance of Mixed Local Preference is a pair $(\langle x, t_h, v_i \rangle, \langle x, t_k, v_j \rangle)$, where x is a Fragment or relation instance, v_i is a value of Value Type t_h, and v_j is a value of Value Type t_k.

Relation

Mixed Local Preference (r.pref.mloc.c)

Domain & Dimension

r.pref.mloc.c $\subseteq \langle x, t^{(m)}, v^{(n)} \rangle$, where c is a Criterion, and $\langle x, t^{(m)}, v^{(n)} \rangle$ is a set of n value assignments on the same Fragment or relation instance x, of values

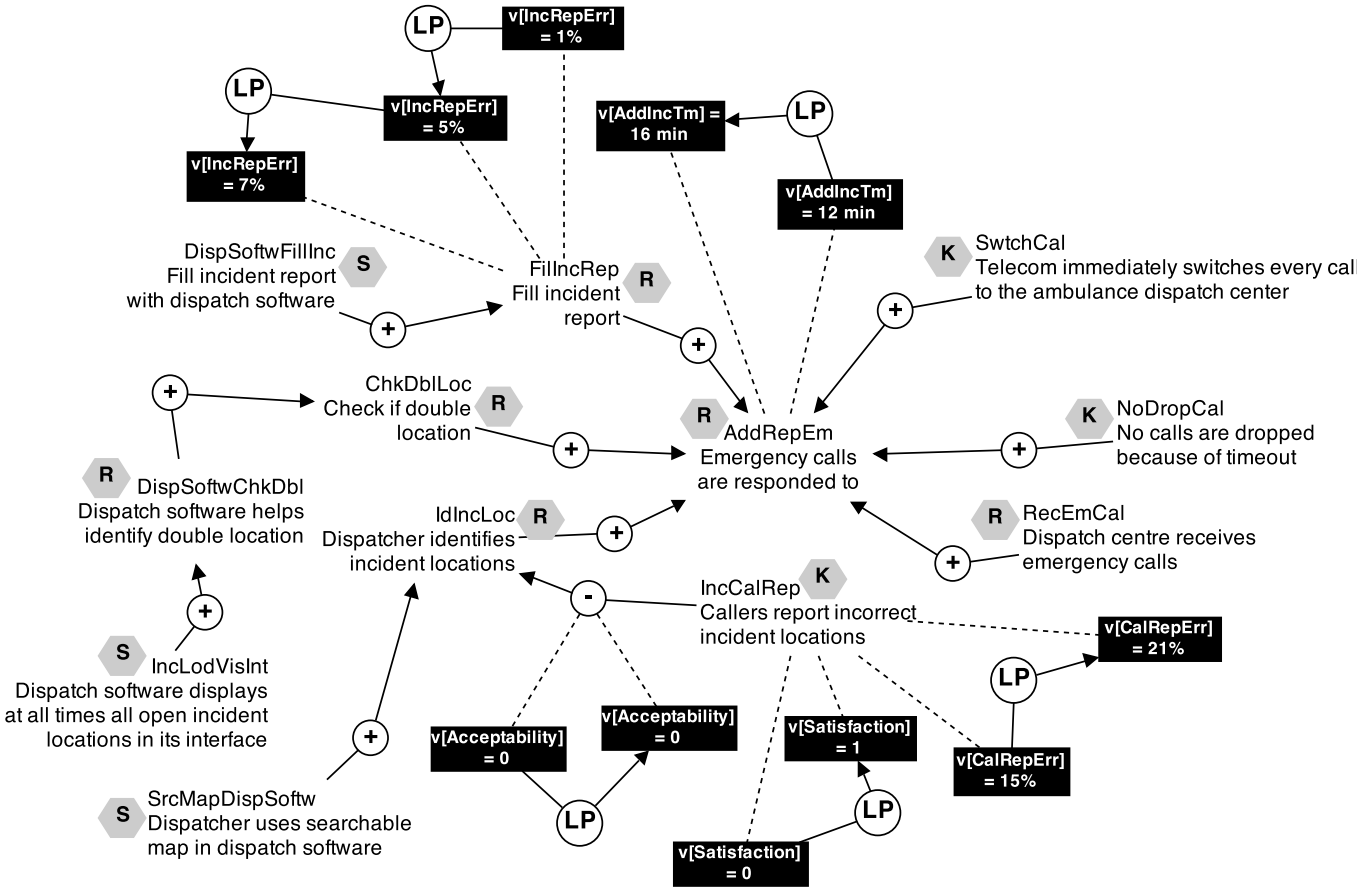


Figure 20: Local Preferences in a model in L.Bellatrix.

from $v^{(n)}$, each of which is one of one of the Value Types in $t^{(m)}$.

Properties

$(\langle x, t_h, v_i\rangle, \langle x, t_k, v_j\rangle \in$ r.pref.mloc.c if $\langle x, t_h, v_i\rangle$ is strictly more desirable to $\langle x, t_k, v_j\rangle$ on the Criterion c.

Reading

$(\langle x, t_h, v_i\rangle, \langle x, t_k, v_j\rangle \in$ r.pref.mloc.c reads "$\langle x, t_h, v_i\rangle$ is strictly preferred to $\langle x, t_k, v_j\rangle$ on the Criterion c".

Language Services

- **s.IsMixLocPref**: Is $\langle x, t_h, v_i\rangle$ is strictly preferred to $\langle x, t_k, v_j\rangle$ on the Criterion c? : Yes, if $(\langle x, t_h, v_i\rangle, \langle x, t_k, v_j\rangle \in$ r.pref.mloc.c.

In the example below, I replace Local Preferences with Mixed Local Preferences in L.Bellatrix. This gives the new language L.Elnath, which can still represent all that L.Bellatrix can.

Exercise 29: Relationship between Local Preferences and Mixed Local Preferences

Can a language that represents Mixed Local Preferences also represent Local Preferences?

Example 11.2. The only difference between L.Elnath below and L.Bellatrix, is that the latter cannot represent preference relation instances over value assignments in which the values are of different Value Types.

Language

Elnath

Language Modules

F, **T**, **V**, **C**, r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat, f.sat.leaf, r.pref.mloc.c

Domain

Same as L.Bellatrix, except that

$$\text{r.pref.mloc.c} \subseteq \mathbf{V} \times \mathbf{V}.$$

Syntax

Same as L.Bellatrix.

Mapping

Same as L.Bellatrix, except that

$$\mathscr{D}(\theta) \in \bigcup_{c \in \mathbf{C}} \text{r.pref.mloc.c.}$$

Language Services

Those of relations and functions in the language.

Figure 21 shows two instances of r.pref.mloc:

$$\begin{array}{lll} (\langle \text{IncCalRep}, \text{v.Satisfaction}, 0\rangle, & & \\ \langle \text{IncCalRep}, \text{v.CallRepEr}, 15\%\rangle) & \in & \text{r.pref.mloc.CalRepPrecLoc} \\ (\langle \text{IncCalRep}, \text{v.Satisfaction}, 0\rangle, & & \\ \langle \text{IncCalRep}, \text{v.CallRepEr}, 21\%\rangle) & \in & \text{r.pref.mloc.CalRepPrecLoc} \end{array}$$

The preferences say that it is strictly preferred to observe that IncCalRep is not satisfied, than to observe 15% or 21% of calls in which callers report incorrect incident locations. The Criterion is crit.CalRepPrecLoc, which is that it is better to observe that callers do not report incident locations incorrectly, rather than observe that there are errors in callers' reports of incident locations. •

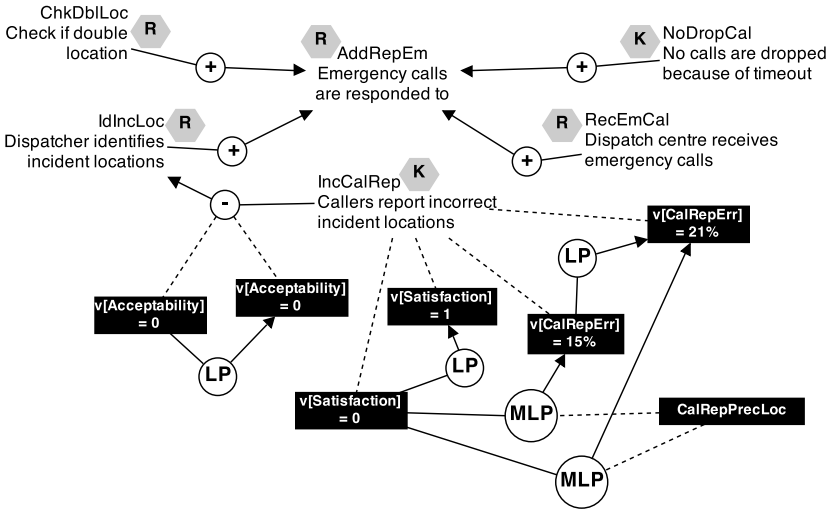


Figure 21: Local Preferences and Mixed Local Preferences in a model in L.Elnath.

The issue with Mixed Local Preferences is that they are over *different* Value Types. The same applies to Mixed Bridge Preferences discussed in Section 11.5. r.pref.mloc has no constraints which would exclude the comparison over some specific pairs of Value Types. This can cause confusion.

For illustration, consider the first Mixed Local Preference in the example above, where the Alternative

$$\langle \text{IncCalRep}, \text{v.Satisfaction}, 0 \rangle$$

is strictly preferred to the Alternative

$$\langle \text{IncCalRep}, \text{v.CallRepEr}, 15\% \rangle.$$

That is, it is strictly preferred that IncCalRep is not satisfied, to observing 15% of calls to report incorrect incident locations. When IncCalRep is not satisfied, does this also means that there are no incorrect incident calls reported at all? Or, does it mean that there is some percentage, but below a tolerable threshold, such as, say 5%? The model does not include information which answers these questions. There is crit.CalRepPrecLoc, but it is also silent on this. There could be a function, which would make the satisfaction of IncCalRep depend on the percentage of incorrect locations reported, or the other way around.

Even if the Criteria are independent, it is still not clear in the model if they are independent. It is up to the modeller to avoid such issues, unless the language comes with a predefined set of Value Types and functions which relate the values over different Value Types. I will return to this later, in Section 11.6 on Criteria.

11.4 Why Bridge Preferences?

Bridge Preferences are preferences over values of the same Value Type, assigned to *different Fragments or relation instances*. Every instance of Bridge Preference is a pair $(\langle x, t, v_1 \rangle, \langle y, t, v_2 \rangle)$, where x and y are two different Fragments or relation instances, and v_1 and v_2 are of the same Value Type t.

Relation
Bridge Preference (r.pref.br.c)
Domain & Dimension
r.pref.br.c $\subseteq \langle x^{(m)}, t, v^{(n)} \rangle$, where c is a Criterion, and $\langle x^{(m)}, t, v^{(n)} \rangle$ is a set of value assignments on different Fragments and, or relation instances from

$x^{(m)}$, and on each one or more assignments of a value from $v^{(n)}$, all of the same Value Type t.

Properties

$(\langle x_h, t, v_i\rangle, \langle x_k, t, v_j\rangle \in$ r.pref.br.c if $\langle x_h, t, v_i\rangle$ is strictly more desirable to $\langle x_k, t, v_j\rangle$ on the Criterion c.

Reading

$(\langle x_h, t, v_i\rangle, \langle x_k, t, v_j\rangle \in$ r.pref.br.c reads "$\langle x_h, t, v_i\rangle$ is strictly preferred to $\langle x_k, t, v_j\rangle$ on the Criterion c".

Language Services

- **s.IsBrPref**: Is $\langle x_h, t, v_i\rangle$ is strictly preferred to $\langle x_k, t, v_j\rangle$ on the Criterion c? : Yes, if $(\langle x_h, t, v_i\rangle, \langle x_k, t, v_j\rangle \in$ r.pref.br.c.

Example 11.3. Figure 22 is a visualisation of a model in L.AInilam, a language made by allowing Bridge Preferences and r.xor instances in models of L.Bellatrix. I define the language as follows.

Language

AInilam

Language Modules

F, **T**, **V**, **C**, r.inf.pos, r.inf.neg, r.xor, f.map.abrel.g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat.x, f.sat.leaf, r.pref.br.c

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, F = c.r∪c.k∪c.s and c.r∩c.k∩c.s = ∅. Influences are over Fragments, r.inf.pos ⊆ F × F, r.inf.neg ⊆ F × F. r.xor instances are over positive and influence relations of the same type

$$\text{r.xor} \in (\text{r.inf.pos}^n) \times (\text{r.inf.neg}^n).$$

Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\mathbf{V} \subseteq (\text{F} \cup \text{r.inf.pos} \cup \text{r.inf.neg}) \times \mathbf{T} \times \text{v.Satisfaction}$$

where **T** = {v.Satisfaction. For every Criterion, preferences are over value assignments, r.pref.br.c ⊆ **V** × **V**, for every $c \in$ **C**.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{aligned}
\alpha &::= x \mid y \mid z \mid \ldots \\
\beta &::= r \mid k \mid s \\
\gamma &::= \beta(\alpha) \\
\delta &::= (\gamma, \gamma) \\
\iota &::= (\delta, \delta, \ldots) \\
\epsilon &::= \langle \alpha, \zeta, \eta\rangle \\
\theta &::= (\epsilon, \epsilon) \\
\phi &::= \gamma \mid \delta \mid \epsilon \mid \theta
\end{aligned}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in$ F. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. δ symbols denote positive and negative influence relations. ι symbols denote r.xor instances, $\mathscr{D}(\iota) \in$ r.xor. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in$ **T**. η denotes a value of a Value Type, and as there is one Value Type, then $\mathscr{D}(\eta) \in$ v.Satisfaction. ϵ symbols denote value assignments, $\mathscr{D}(\epsilon) \in$ **V**. θ symbols denote preference relations

$$\mathscr{D}(\theta) \in \bigcup_{c \in \mathbf{C}} \text{r.pref.br.c.}$$

Language Services

Those of relations and functions in the language.

Figure 22 shows one Local Preference and four Bridge Preferences (marked "BP"). The preferences are over assignments of v.Satisfaction values to Fragments and relation instances. The model in the Figure shows two assignments of v.Satisfaction values to Fragments and relation instances. Let $\langle X \cup E, \text{v.Satisfaction}, W_1 \rangle$ denote one such assignment, where X is the set of all Fragments, and E the set of all influence relation instances. Let $\langle X \cup E, \text{v.Satisfaction}, W_2 \rangle$ denote the other assignment. Values assigned according to $\langle X \cup E, \text{v.Satisfaction}, W_1 \rangle$ are shown on black rectangles and squares. Those of $\langle X \cup E, \text{v.Satisfaction}, W_2 \rangle$ are on grey rectangles and squares.

There is one Local Preference in the Figure, which in L.Alnilam amounts to a Bridge Preference which is over values of the same variable (that is, over value assignments to the same Fragment, of values of the same Value Type):

$$(\langle \text{AmbArrIncLoc}, \text{v.Satisfaction}, 1 \rangle, \langle \text{AmbArrIncLoc}, \text{v.Satisfaction}, 0 \rangle) \in \text{r.pref.br.SatImpReq}$$

It says that it is strictly preferred to satisfy AmbArrIncLoc, than to fail to satisfy it. crit.SatImpReq abbreviates that important requirements should be satisfied, so the Local Preference says that AmbArrIncLoc is an important requirement.

As there are r.xor instances in the model in the Figure, the Local Preference above is not enough to help me choose among the Preference Alternatives shown. The four Bridge Preferences compare Preference Alternatives are as follows. The first two are over Fragments, the last two over instances of r.inf.pos. All are associated to crit.MoreDecSup, which is that there should be more technology in the system to support decision-making.

$$\{(\langle \text{AutAmbList}, \text{v.Satisfaction}, 1 \rangle, \langle \text{ManTrckAmb}, \text{v.Satisfaction}, 1 \rangle),$$
$$(\langle \text{UpdAutoAmbList}, \text{v.Satisfaction}, 1 \rangle, \langle \text{ManTrckAmb}, \text{v.Satisfaction}, 1 \rangle),$$
$$(\langle (\text{DispSoftwRnkAmb}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle,$$
$$\langle (\text{NoAmbRecomm}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle),$$
$$(\langle (\text{DispAmbRnk}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle,$$
$$\langle (\text{NoAutAmbRnk}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle)$$
$$\} \subseteq \text{r.pref.br.MoreDecSup}$$

According to these Bridge Preferences, $\langle X \cup E, \text{v.Satisfaction}, W_1 \rangle$ is better than $\langle X \cup E, \text{v.Satisfaction}, W_2 \rangle$. This is because all the strictly preferred value assignments are in $\langle X \cup E, \text{v.Satisfaction}, W_1 \rangle$.

The conclusion would not have been as clear, if at least one of the Bridge Preferences above was reversed. For example, if

$$(\langle (\text{DispSoftwRnkAmb}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle, \langle (\text{NoAmbRecomm}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle) \in \text{r.pref.br.MoreDecSup}$$

is replaced by

$$(\langle (\text{NoAmbRecomm}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle, \langle (\text{DispSoftwRnkAmb}, \text{ChoAmb}), \text{v.Satisfaction}, 1 \rangle) \in \text{r.pref.br.MoreDecSup}$$

Also, the Bridge Preferences in Figure 22 suggest that Combination with AutoAmbList, UpdAutoAmbList, DispSoftwRnkAmb, and DispAmbRnk are better than other Combinations. Recall from Figure 14 that there are four Combinations according to the r.xor instances in the model. The first Combination, shown in Figure 14(a) is the only one of the four, which includes all Fragments which appear in the Bridge Preferences. •

11.5 Why and How to Use Mixed Bridge Preferences?

Mixed Bridge Preferences are preferences over values of *different Value Types*, assigned to *different Fragments or relation instances*. Every instance of Mixed Bridge Preference is a pair $(\langle x, t_1, v_1 \rangle, \langle y, t_2, v_2 \rangle)$, where x and y are two different Fragments or relation instances, t_1 and t_2 two different Value Types, and v_1 and v_2 are, respectively, values of t_1 and of t_2.

Relation

Mixed Bridge Preference (r.pref.mbr.c)

Domain & Dimension

r.pref.mbr.c $\subseteq \langle x^{(m)}, t^{(u)}, v^{(n)} \rangle$, where c is a Criterion, and $\langle x^{(m)}, t^{(u)}, v^{(n)} \rangle$ is a set of n value assignments $\langle x, t, v_i \rangle$ on different Fragments and, or relation instances from $x^{(m)}$, and on each one or more assignments of a value from $v^{(n)}$, each of a Value Type from $t^{(u)}$.

Properties

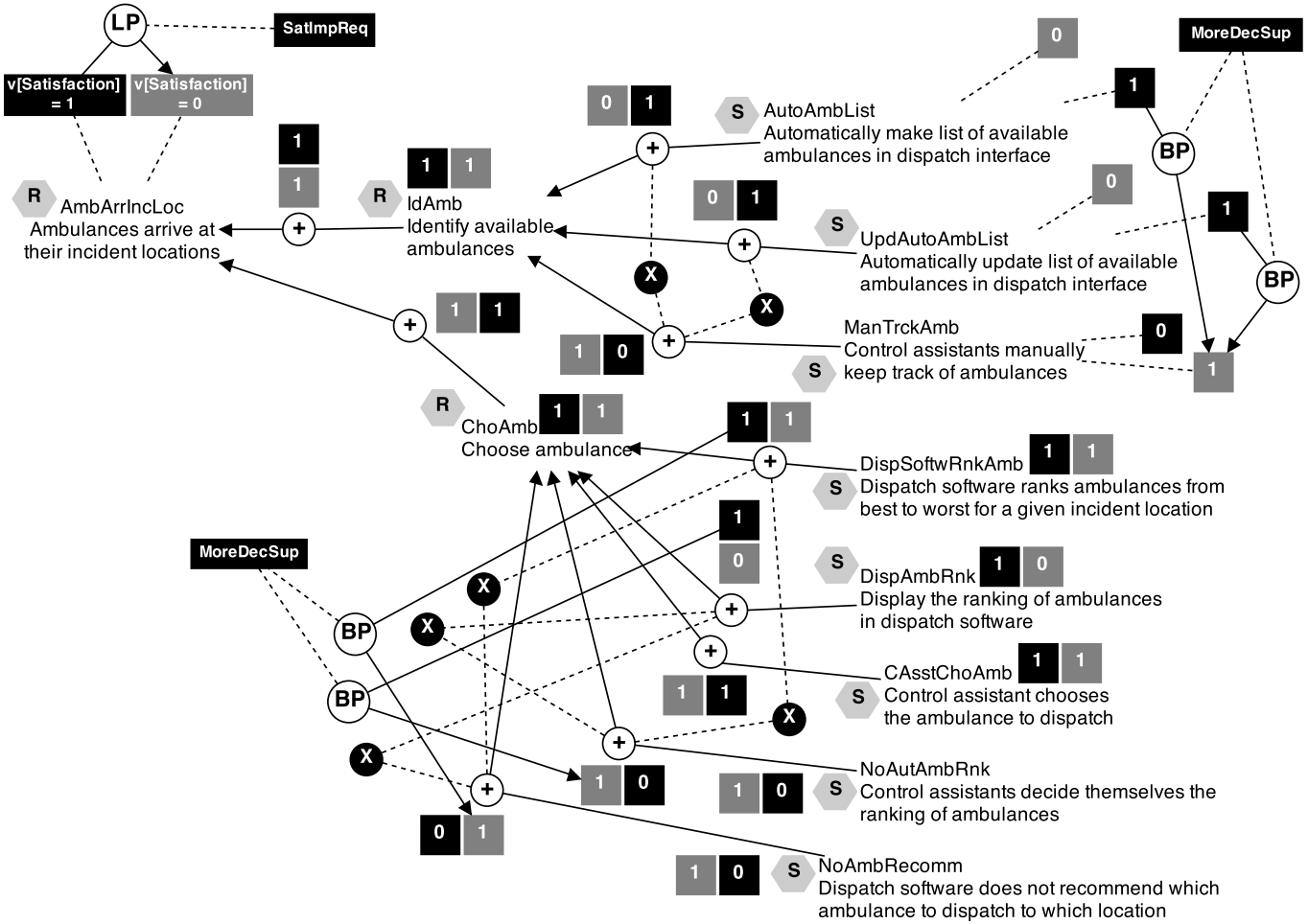


Figure 22: Local Preferences and Bridge Preferences.

$(\langle x_h, t_g, v_i\rangle, \langle x_k, t_l, v_j\rangle \in$ r.pref.mbr.c if $\langle x_h, t_g, v_i\rangle$ is strictly more desirable to $\langle x_k, t_l, v_j\rangle$ on the Criterion c.

Reading

$(\langle x_h, t_g, v_i\rangle, \langle x_k, t_l, v_j\rangle \in$ r.pref.mbr.c reads "$\langle x_h, t_g, v_i\rangle$ is strictly preferred to $\langle x_k, t_l, v_j\rangle$ on the Criterion c".

Language Services

- **s.IsMBrPref**: Is $\langle x_h, t_g, v_i\rangle$ is strictly preferred to $\langle x_k, t_l, v_j\rangle$ on the Criterion c? : Yes, if $(\langle x_h, t_g, v_i\rangle, \langle x_k, t_l, v_j\rangle \in$ r.pref.mbr.c.

Example 11.4. Figure 23 is a visualisation of a model in L.Canopus, which is made by allowing Mixed Bridge Preferences in the models of L.Elnath.

Language

Canopus

Language Modules

F, **T**, **V**, **C**, r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat, f.sat.leaf, r.pref.mbr.c

Domain

Same as L.Elnath, except that

$$\text{r.pref.mbr.c} \subseteq \mathbf{V} \times \mathbf{V}.$$

Syntax

Same as L.Elnath.

Mapping

Same as L.Elnath, except that

$$\mathscr{D}(\theta) \in \bigcup_{c \in \mathbf{C}} \text{r.pref.mbr.c}.$$

Language Services

Those of relations and functions in the language.

The figure includes the following three r.pref.mbr instances:

$$\begin{aligned}
&(\langle \text{AddRepEm}, \text{v.AddIncTm}, 12\text{min}\rangle,\\
&\qquad \langle \text{FillIncRep}, \text{v.IncRepErr}, 1\%\rangle) \quad \in \quad \text{r.pref.mbr}\\
&(\langle(\text{IncCalRep}, \text{IdIncLoc}), \text{v.Acceptability}, 0\rangle,\\
&\qquad \langle \text{IncCalRep}, \text{v.Satisfaction}, 0\rangle) \quad \in \quad \text{r.pref.mbr}\\
&\qquad (\langle \text{IncCalRep}, \text{v.Satisfaction}, 0\rangle,\\
&\langle(\text{IncCalRep}, \text{IdIncLoc}), \text{v.Acceptability}, 0\rangle) \quad \in \quad \text{r.pref.mbr}
\end{aligned}$$

The first is the strict preference of observing 12 minutes as the average time to respond to incident, to the 1% error rate in incident reports. It is due to crit.QuickResponseOverRepErr, which is that it is better to lower average time to address an incident, than to lower the error rate in incident reports.

The second is a strict preference of not accepting that IncCalRep negatively influences IdIncLoc, to having that IncCalRep is not satisfied. In other words, it is strictly preferred not to even consider if IncCalRep is satisfied, to having it fail.

The third, and final Mixed Bridge Preference is that it is strictly preferred to have IncCalRep not satisfied, than to accept that IncCalRep negatively influences IdIncLoc.

The second and third are due to crit.CalRepPrecLoc, which is that it is better to observe that callers do not report incident locations incorrectly, rather than observe that there are errors in callers' reports of incident locations. •

11.6 Where to Find Criteria in Requirements?

There are not enough preference relation instances in Figure 20 to create a total order, even over a single Criterion. For example, I know from the model that the 12min average time to respond to incidents is strictly preferred to 16min, but is 15min also

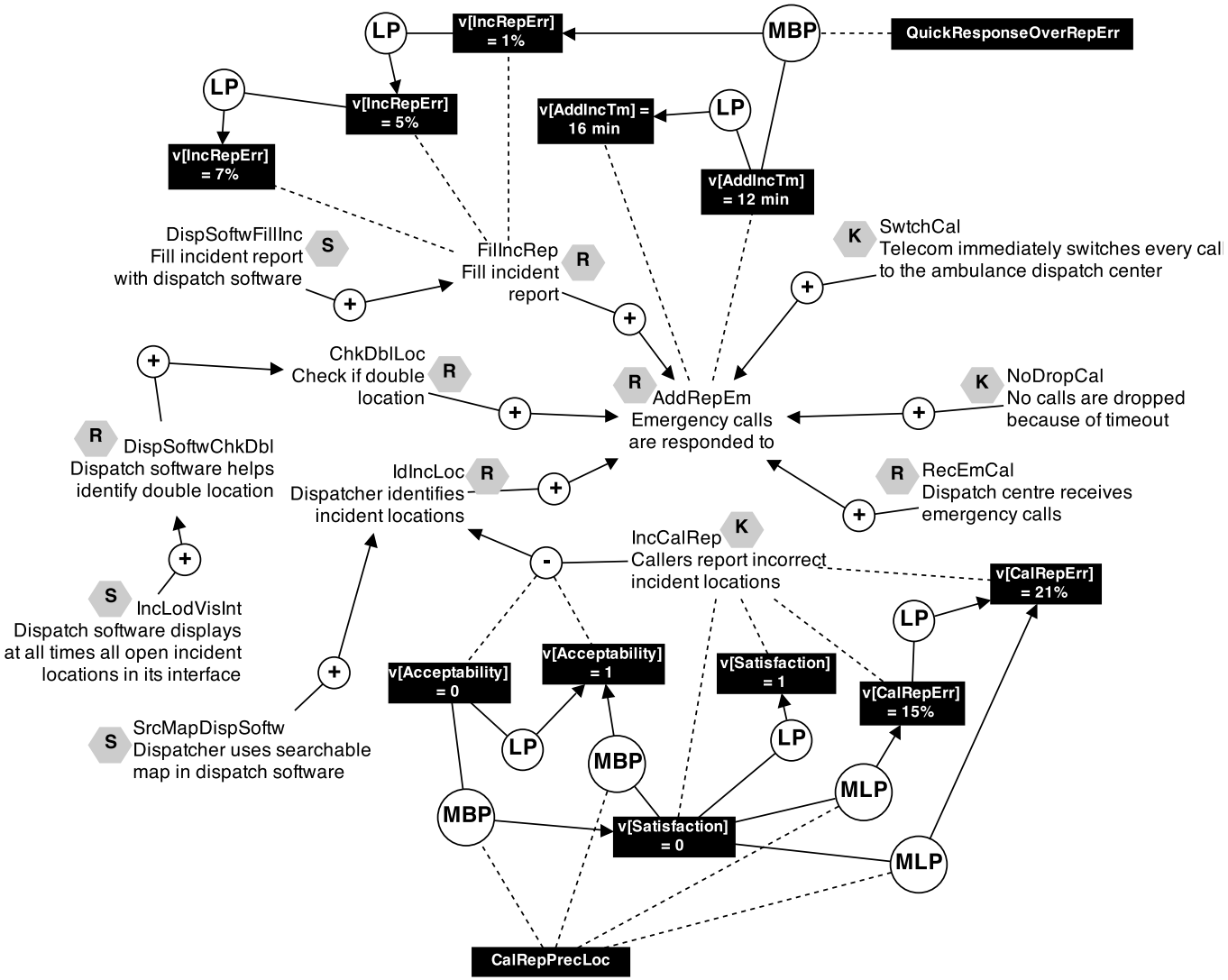


Figure 23: Local Preferences, Mixed Local Preferences, and Mixed Bridge Preferences.

preferred to 16min, and is it less desirable than 12min? It may seem obvious that I should prefer lower average time, but do stakeholders who matter? In any case, the model shows few preferences.

If you want to elicit if 12min incident response time is more preferred than 13min, 14min, or 15min, or all of them, you could give many such pairs, and ask stakeholders for their opinion. A quicker approach may be to look, in Fragments, for simpler statements which suggest many preference instances. For example, suppose that stakeholders agree that they prefer lower to higher incident response times, and that there is no lower limit to how short the average time can be. Perhaps a stakeholder said that she wants low incident response times, and the others agreed.

In RE, statements such as "low incident response times", "less maintenance", "low cost" are not uncommon, and are usually called nonfunctional requirements [14], or softgoals [98].

One way to use these statements is to view them as revealing Criteria. For example, "quickly respond to emergency calls" suggests a Criterion. Denote it crit.AddIncTm. You might then decide to measure the speed to respond to emergency calls by the average time to respond to incidents, that is v.AddIncTm. And consequently, when given two value assignments of v.AddIncTm, that Criterion returns a higher value for assignment with the lower average time to respond to emergency calls. In other words, the Criterion returns a preference over the two value assignments.

Below, I define crit.LowAddIncTm in such a way as to generate preference relation instances over value assignments, when it is given a pair of v.AddIncTm value assignments. The Language Module template for Criteria is the same as for functions. The Criterion takes a pair of value assignments, and returns the preference relation over them, and in this sense amounts to a function. Nevertheless, I want to distinguish Criteria from other kinds of functions, hence the dedicated template.

Criterion

Prefer low time to address incidents (crit.low.AddIncTm)

Input

A pair of value assignments $\langle x_1, \text{v.AddIncTm}, v_1\rangle$ and $\langle x_2, \text{v.AddIncTm}, v_2\rangle$.

Do

Let v_i be the minimum and v_j the maximum in $\{v_1, v_2\}$, and

$w = (\langle x_i, \text{v.AddIncTm}, v_i\rangle, \langle x_j, \text{v.AddIncTm}, v_j\rangle) \in$ r.pref.mloc.low.AddIncTm.

Output

w.

Language Services

- **s.WhLowAddIncTm**: According to crit.LowAddIncTm, which of the value assignments $\langle x_1, \text{v.AddIncTm}, v_1\rangle$ and $\langle x_2, \text{v.AddIncTm}, v_2\rangle$ is strictly preferred to the other? : The one strictly preferred according to w which this module outputs.

You can define a more general Criterion, which can work with values assignments where Value Types are real numbers, and lowest or highest values are the most desirable. Below is such a module.

Criterion

Prefer higher (or lower) v.t values (crit.d.t)

Input

- A Value Type t,
- a parameter d, which is either $d = low$ or $d = high$, and
- a pair of value assignments $\langle x_1, \text{v.t}, v_1\rangle$ and $\langle x_2, \text{v.t}, v_2\rangle$, such that v.t takes real values.

Do

Let v_i be the minimum and v_j the maximum in $\{v_1, v_2\}$, and

- if $d = Low$, then $w = (\langle x_i, \text{v.t}, v_i\rangle, \langle x_j, \text{v.t}, v_j\rangle) \in$ r.pref.mloc.d.t, else

- if $d = High$, then $w = (\langle x_j, \text{v.t}, v_j\rangle, \langle x_i, \text{v.t}, v_i\rangle) \in$ r.pref.mloc.d.t.

Output

w.

Language Services

- **s.WhPref.t**: According to crit.d.t, which of the value assignments $\langle x_1, t, v_1\rangle$ and $\langle x_2, t, v_2\rangle$ is strictly preferred to the other? : The one strictly preferred according to w which this module outputs.

If a model has no Criteria which generate preference relations, then all individual preferences in the model need to come from some other approach to preference elicitation. Otherwise, if you have Criteria which do generate preference relation instances, and there are value assignments which these Criteria apply to, then you can automatically add preference relation instances. The following example illustrates this.

Example 11.5. Figure 24 shows a model in L.Bellatrix, before and after adding two Criteria to it. In Figure 24(a), only value assignments are shown, and no preferences.

Figure 24(b) shows the result of adding crit.low.IncRepEr and crit.low.AddIncTm to the model in Figure 24(a). Doing so adds preference relations over value assignments.

Adding crit.low.IncRepEr resulted in adding two Local Preferences, to say, respectively, that 1% error rate in incident reports is strictly preferred to a 5%, and that 5% is strictly preferred to 7%.

Adding crit.low.AddIncTm results in the strict preference for 12min average time to address an incident, to 16min average time to address an incident. •

Criteria can reflect more complicated preferences than crit.d.t. The following example is an illustration. It defines a Criterion, which is remotely related to a classical proposal in the field of multiple-criteria decision analysis.

Example 11.6. Suppose that there is a Value Type v.ImplCost, and that you assign its values to Fragments, to indicate an estimate of the cost to implement what the Fragment describes.

Moreover, suppose that you have elicited the following statement, or concluded this from having elicited some other information from stakeholders: "The lowest implementation cost Alternative is best, unless it is not less than 20% cheaper than the next lowest cost Alternative, in which case the latter is better than the former".

This is inspired by the so-called Type V criterion in the PROMETHEE approach to multiple-criteria decision analysis [17], where the individual is assumed to be indifferent to value assignments, until the difference between them reaches a certain value. Here, I adapt this idea to there being no indifference relation, and consider that there is a strict preference, until the difference between the two assigned values goes above a threshold, which is some given percentage of the higher value. If the value goes above the threshold, then the strict preference reverses. The following Criterion captures these ideas.

Criterion

Prefer lower of two v.t values, until their difference is more than h% of the higher (crit.low.rev.h)

Input

- v.t, which must be a subset of real numbers,
- a percentage value h%, and
- a pair of value assignments $\langle x_1, \text{v.t}, v_1\rangle$ and $\langle x_2, \text{v.t}, v_2\rangle$.

Do

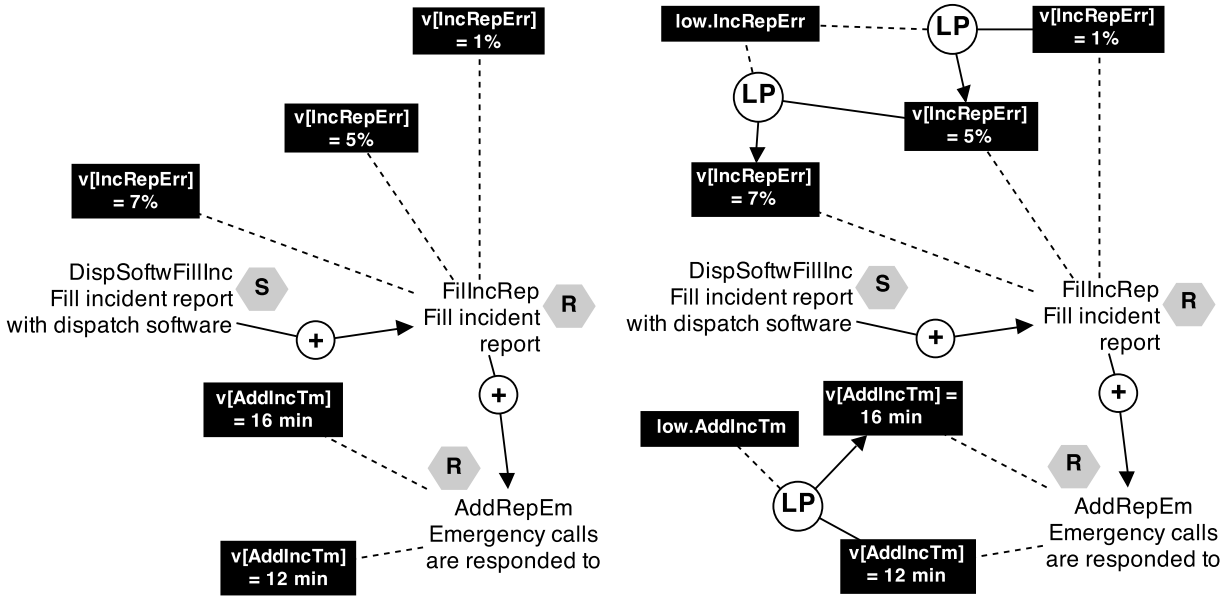
Let v_i be the minimum and v_j the maximum in $\{v_1, v_2\}$, and if $|v_i - v_j|/v_j > h/100$, then

$w = (\langle x_i, \text{v.t}, v_i\rangle, \langle x_j, \text{v.t}, v_j\rangle) \in$ r.pref.mloc.rev.d.t, else

$w = (\langle x_j, \text{v.t}, v_j\rangle, \langle x_i, \text{v.t}, v_i\rangle) \in$ r.pref.mloc.rev.d.t.

Output

w.



(a) Before new Criteria.

(b) After new Criteria.

Figure 24: A model before and after adding two Criteria.

Language Services

- **s.WhPref.low.rev.h**: According to crit.low.rev.h, which of the value assignments $\langle x_1, \text{v.t}, v_1 \rangle$ and $\langle x_2, \text{v.t}, v_2 \rangle$ is strictly preferred to the other? : The one strictly preferred according to w which this module outputs.

•

The more general point is that preference relation instances can be automatically added to a model, in case you have defined a Criterion which suggests such preferences. There are many proposals for generic Criteria which can be used in this way, especially in the field of multiple-criteria decision analysis [141, 94, 45]. The issue which remains unsolved is how to make sure that the Criteria do correspond to stakeholders' preferences, an issue to be solved via elicitation, validation, and negotiation, rather than, unfortunately, Language Modules.

11.7 How to Find a Better and the Best Outcome?

How would you deliver the following Language Services?

- **s.BestOutcome**: Which is the best Outcome in model M?
- **s.BetterOutcome**: Which of the two Outcomes o_i and o_j is better in model M?

Both are problems of preference aggregation. There are various proposals for how to do preference aggregation [16, 103, 45].

Given a model M with value assignments, preference relations, and Criteria, I want to deliver s.BestOutcome and s.BetterOutcome by mapping the variables and preferences to a Conditional Preference Network (CP-Net) [16]. There are known algorithms for CP-Nets, which can be used to deliver both s.BestOutcome and s.BetterOutcome.

To reduce the number of preference relation instances that need to be elicited, CP-Nets use conditional preference relations. A conditional preference is a pair (a, p), where p is a preference relation instance and $a = \langle x, \text{v.t}, v \rangle$ is a value assignment. The idea is that if the Outcome includes a, then the preference p should be taken into account when computing answers to s.BestOutcome and s.BetterOutcome. In order to map models to CP-Nets, it should be possible to represent conditions of preferences in models. This is done with the relation r.pref.cond below.

Relation

Conditional preference (r.pref.cond)

Domain & Dimension

r.pref.cond $\subseteq V \times R$, where V is a set of value assignments, and R is a set of Mixed Bridge Preference relation instances.

Properties

If the preference $p \in P$ should be taken into account when comparing all Outcomes which include $\langle x, \text{v.t}, v \rangle$, then let

$$(\langle x, \text{v.t}, v \rangle, p) \in \text{r.pref.cond}$$

Reading

$(\langle x, t, v \rangle, p) \in$ r.pref.cond reads "use the preference relation p when comparing Outcomes, only if at least one of the Outcomes s includes $\langle x, t, v \rangle$".

Language Services

- **s.IsCondPref**: Should the preference p be used to compare Outcomes in the set O? : Yes, if there is $(\langle x, \text{v.t}, v \rangle, p) \in$ r.pref.cond and all Outcomes in O include $\langle x, t, v \rangle$.

The following language allows the representation of r.pref.cond instances, r.pref.mbr.c instances, over satisfaction and approval values, in models with positive and negative influence relations over Fragments.

Language

Procyon

Language Modules

F, **T**, **V**, **C**, r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, f.sat.inf.pos, f.sat.inf.neg, f.sat, f.sat.leaf, r.pref.mbr.c, r.pref.cond

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, F = c.r∪c.k∪c.s and c.r∩c.k∩c.s = ∅. Influences are over Fragments, r.inf.pos ⊆ F × F, r.inf.neg ⊆ F × F. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\mathbf{V} \subseteq (\mathsf{F} \cup \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg}) \times \mathbf{T} \times (\mathsf{v.Satisfaction} \cup \mathsf{v.Approval})$$

where **T** = {v.Satisfaction, v.Approval}, with v.Satisfaction = {0,1} and v.Approval = {0,1}. For every Criterion, preferences are over value assignments, r.pref.mbr.c ⊆ **V** × **V**, for every $c \in \mathbf{C}$. Conditional preferences are over value assignments and non-conditional preference relation instances,

$$\mathsf{r.pref.cond} \subseteq \mathbf{V} \times \bigcup_{c \in \mathbf{C}} \mathsf{r.pref.mbr.c}.$$

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{aligned}
\alpha &::= x \mid y \mid z \mid \ldots \\
\beta &::= r \mid k \mid s \\
\gamma &::= \beta(\alpha) \\
\delta &::= (\gamma, \gamma) \\
\epsilon &::= \langle \alpha, \zeta, \eta \rangle \\
\theta &::= (\epsilon, \epsilon) \\
\iota &::= (\epsilon, \theta) \\
\phi &::= \gamma \mid \delta \mid \epsilon \mid \theta
\end{aligned}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in \mathsf{F}$. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in \mathsf{c.r}$, $\mathscr{D}(k(\alpha)) \in \mathsf{c.k}$, $\mathscr{D}(s(\alpha)) \in \mathsf{c.s}$. δ symbols denote positive and negative influence relations. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in \mathbf{T}$. η denotes a value of a Value Type. ϵ symbols denote value assignments, $\mathscr{D}(\epsilon) \in \mathbf{V}$. θ symbols denote non-conditional preference relations

$$\mathscr{D}(\theta) \in \bigcup_{c \in \mathbf{C}} \mathsf{r.pref.loc.c},$$

and ι symbols denote conditional preference relations, $\mathscr{D}(\iota) \in \mathsf{r.pref.cond}$.

Language Services

Those of relations and functions in the language.

Given a model M in L.Procyon, I need the following tuple from it:

$$(\mathbf{V}(M), \langle x^{(m)}, t^{(u)}, v^{(n)} \rangle, \mathsf{r.pref.mbr}, \mathsf{r.pref.cond})$$

where $\langle x^{(m)}, t^{(u)}, v^{(n)} \rangle$ is a set of value assignments, as explained in Section 11.5, and

$$\begin{aligned}
\mathsf{r.pref.mbr.c} &\subseteq \langle x^{(m)}, t^{(u)}, v^{(n)} \rangle \times \langle x^{(m)}, t^{(u)}, v^{(n)} \rangle, \\
\mathsf{r.pref.cond} &\subseteq \langle x^{(m)}, t^{(u)}, v^{(n)} \rangle \times \mathsf{r.pref.mbr.c}, \text{ for every } c \in C.
\end{aligned}$$

The following function takes the tuple above, and makes a CP-Net from it.

Function

Make a CPNet
(f.make.CPNet)

Input

$(\mathbf{V}(M), \langle x^{(m)}, t^{(u)}, v^{(n)} \rangle, \mathsf{r.pref.mbr}, \mathsf{r.pref.cond})$.

Do

1. Let $\mathbf{G}(M)$ be a graph, in which $\mathbf{V}(M)$ is the set of nodes, and every node $x.t \in \mathbf{V}(M)$ is annotated with a so-called Conditional Preference Table (CPT), denoted $CPT(x.\text{v.t})$.

2. For every variable $x.\text{v.t} \in \mathbf{V}(M)$, find all Local Preference instances over that variable, let that set be $P(x.\text{v.t})$ and find all conditional preferences to members of $P(x.\text{v.t})$, and let that set be $C_P(x.\text{v.t})$.

3. For every variable $x.\text{v.t} \in \mathbf{V}(M)$, define its $CPT(x.\text{v.t})$ by adding every Mixed Bridge Preference $(\langle x, \text{v.t}, v_i\rangle, \langle x, \text{v.t}, v_j\rangle) \in P(x.t)$ and member of $C_P(x.\text{v.t})$ to the relevant $CPT(x.\text{v.t})$.

4. For each variable $x.\text{v.t} \in \mathbf{V}(M)$, if its $CPT(x.\text{v.t})$ is not complete, then elicit or otherwise find the missing Mixed Bridge Preferences and r.pref.cond instances, and add them to $CPT(x.\text{v.t})$.

Output

The CP-Net $\mathbf{G}(M)$.

Language Services

- s.BestOutcome: The best Outcome is the Outcome returned by an outcome optimisation query [16] on the CP-Net $\mathbf{G}(M)$.
- s.BetterOutcome: The better Outcome is the one returned by a dominance query [16] on the CP-Net $\mathbf{G}(M)$.

Figure 25 shows a model in L.Procyon. There are no value assignments in the Figure. Each Fragment in the Figure is annotated with two Local Preference instances. One gives the preference over satisfaction values, and the other over approval values for that Fragment.

There are four conditional preference relations in the Figure. Two indicate that preference over satisfaction values of ChkDblLoc depends on the satisfaction value of IncCalRep. The other two say that preference over satisfaction values of DispSoftwChkDbl depends on the satisfaction value of ChkDblLoc.

Given these conditional preferences, and all other Local Preferences in the Figure, what is the best Outcome? That is, what are the best assignments of values to all Fragments and relation instances in that model?

To answer this, the first step is to make a CP-Net, so as to find the best value assignment to the Fragments whose satisfaction values involve conditional preferences. The resulting CP-Net is shown in Figure 25(b). Next, running an outcome optimisation query on the CP-Net in Figure 25(b) will result in the graph in Figure 26, where each edge runs from a better to a worse combination of value assignments. That graph shows that the best combination assigns the satisfaction value 0 to IncCalRep, ChkDblLoc, and DispSoftwChkDbl.

While Figure 26 does show the best combination of satisfaction values over three Fragments, the best Outcome will not necessarily include that best combination. The reason is that you still need to assign satisfaction and approval values to all other Fragments and relation instances in the model, and in doing that, you need to take care about how satisfaction values propagate via f.sat.inf.pos, f.sat.inf.neg, f.sat, and f.sat.leaf.

The best approval Outcome is easy to find. As the approval value of a Fragment, or relation instance, is independent of other assignments of approval values in the model, you can assign the preferred approval value to every Fragment and relation instance. To keep the figures simple, I assume that the preferred approval value for every influence relation is 1. The best approval Outcome is shown in Figure 27.

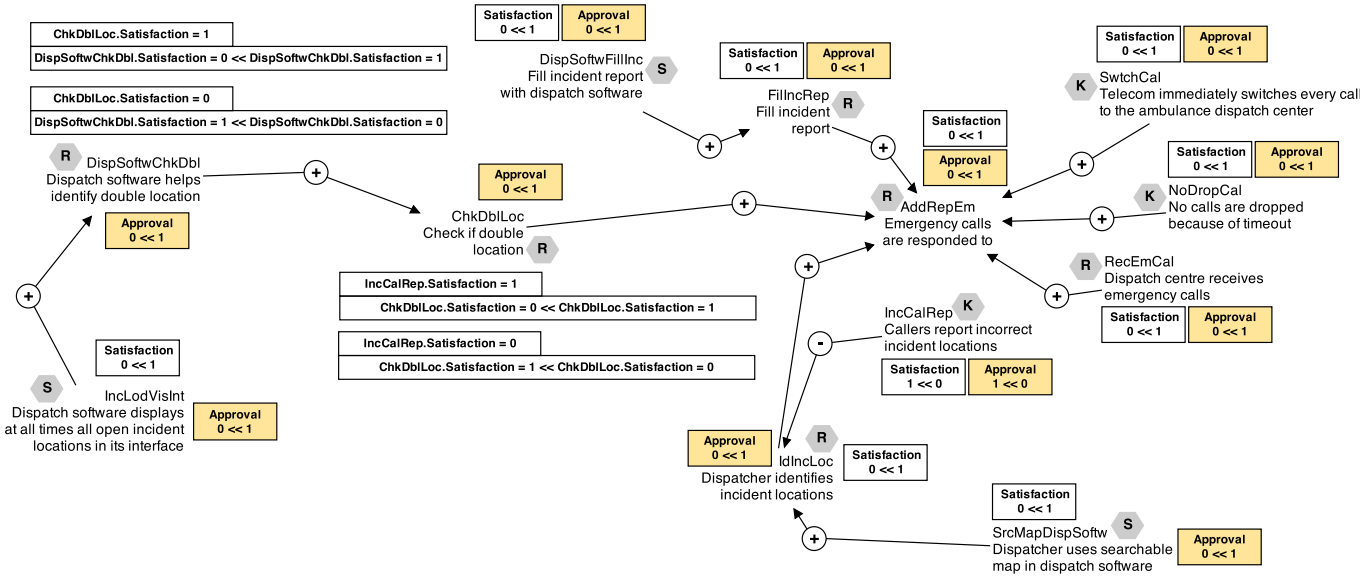
Figure 28 shows an Outcome which includes the best combination of satisfaction values of IncCalRep, ChkDblLoc, and DispSoftwChkDbl, and the best assignment of satisfaction values to other Fragments, which still satisfies propagation rules in f.sat.inf.pos, f.sat.inf.neg, f.sat, and f.sat.leaf. The obvious problem with that Outcome is that AddRepEm is not satisfied.

You can repair AddRepEm by choosing an Outcome which ignores the conditional relations. This Outcome is shown in Figure 29. Another approach is to change the conditional preferences on DispSoftwChkDbl, so that they are conditional on the satisfaction value of AddRepEm, rather than ChkDblLoc. Also, you could change the influence relations, by removing the one from ChkDblLoc to AddRepEm.

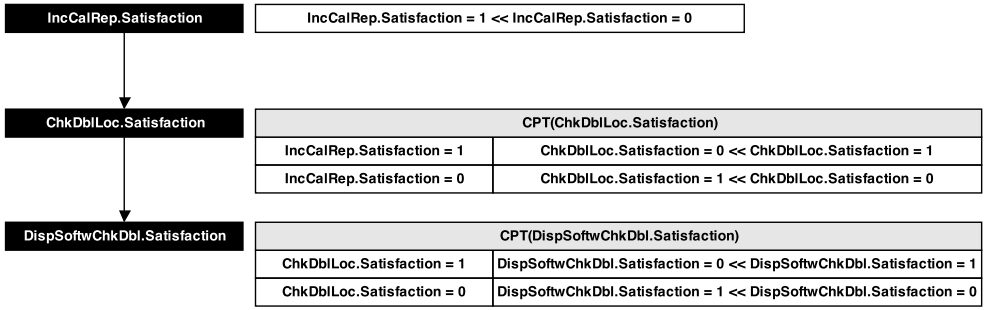
11.8 Summary on Preferences

I introduced simple preference relations, illustrated how to add them to languages and represent them in models, and finally, how to map models to CP-Nets, in order to use conditional preferences to find best Outcomes. Many open questions remain outside the scope of this tutorial:

- How to have conditional preferences which are not over Local Preferences, but over Mixed Bridge Preferences?
- How to represent that preferences are conflicting, which is that improvement over one leads to a decrease over another?



(a) Model in L.Procyon, with no value assignments.



(b) CP-Net made from conditional preferences in Figure 25(a).

Figure 25: Conditional preferences and the corresponding CP-Net.

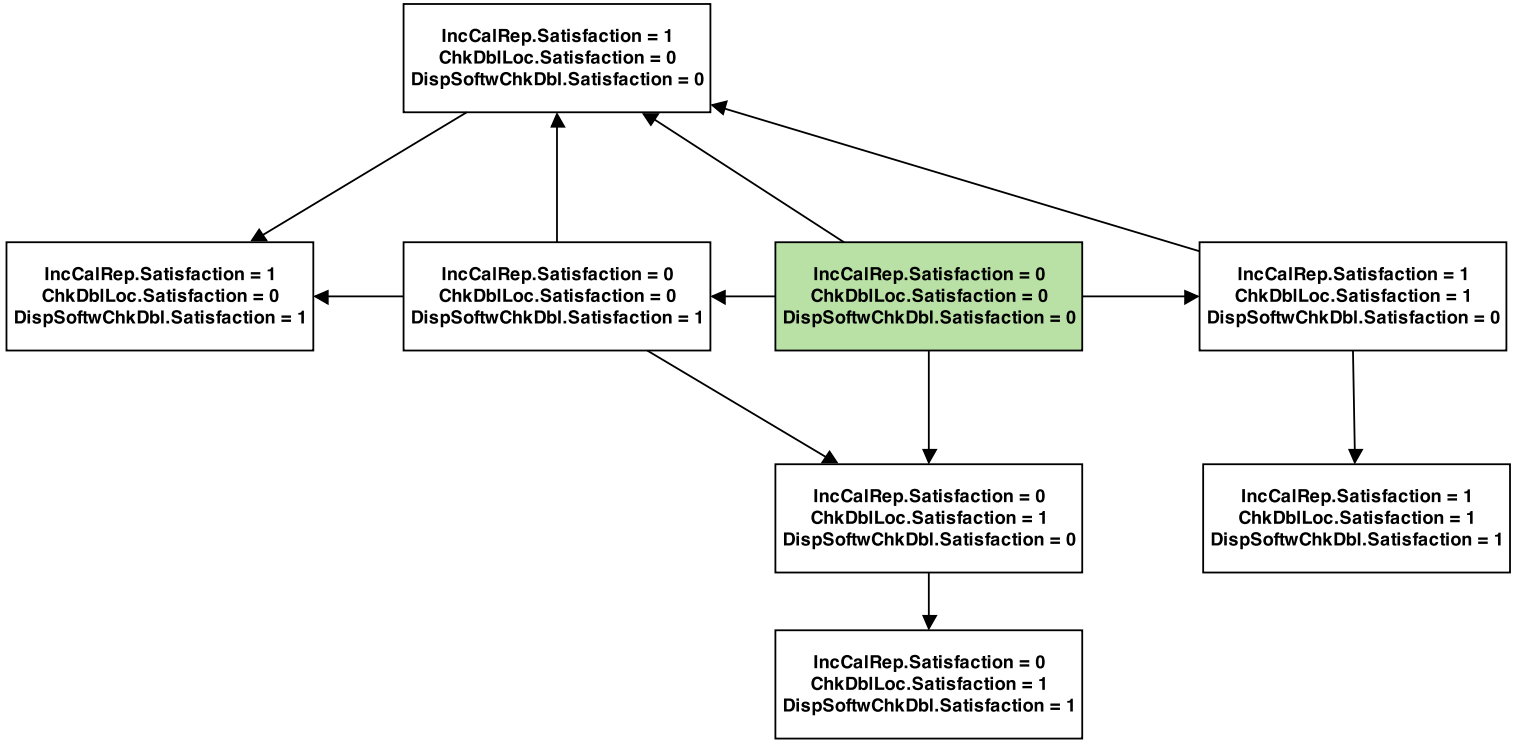


Figure 26: Preference graph induced from the CP-Net in Figure 25(b).

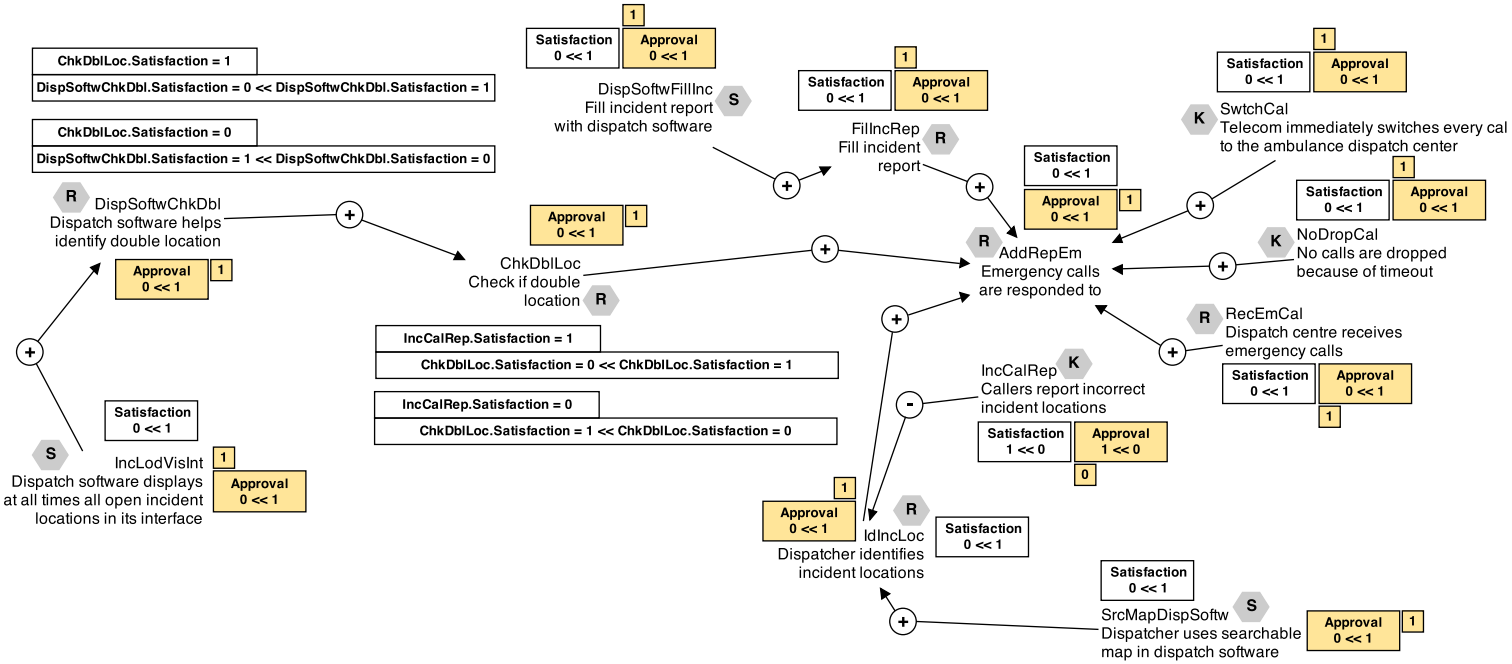


Figure 27: Best approval Outcome, assuming 1 is the preferred approval value on all relation instances.

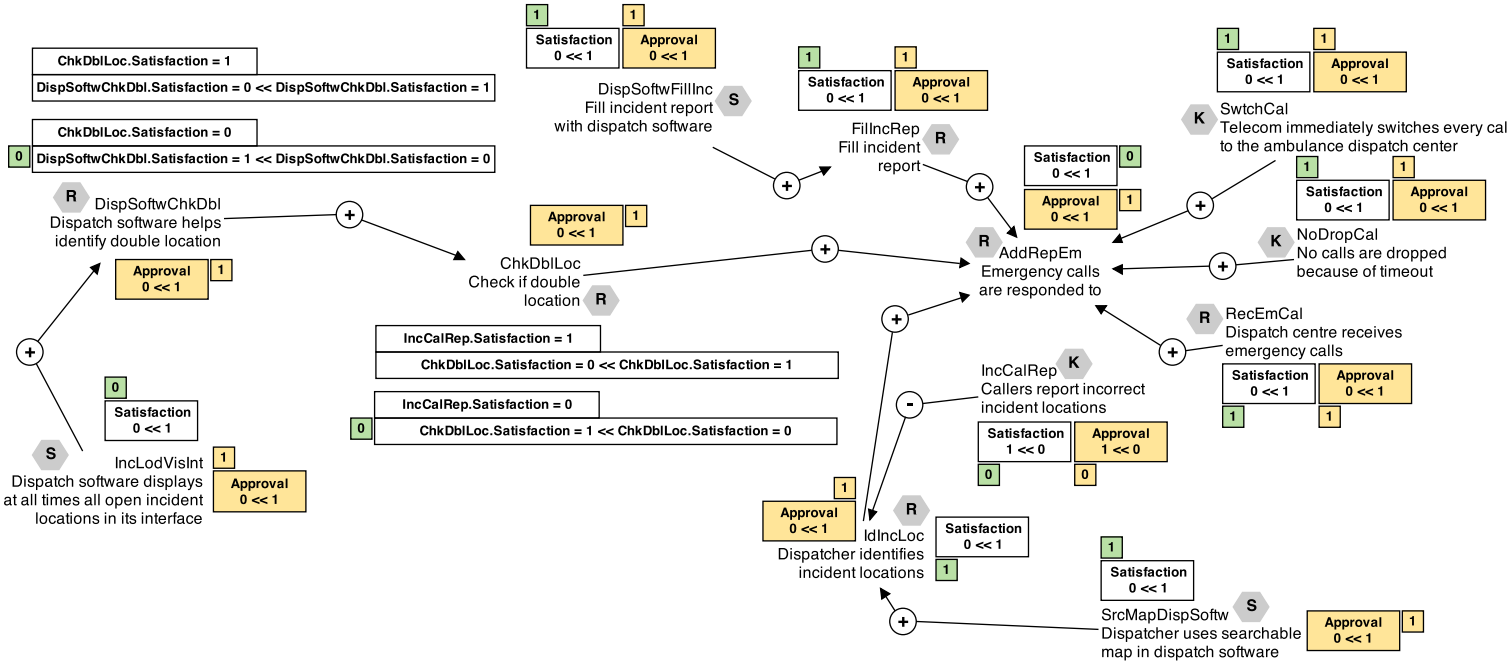


Figure 28: Best Outcome which includes the best combination of satisfaction values according to conditional preferences.

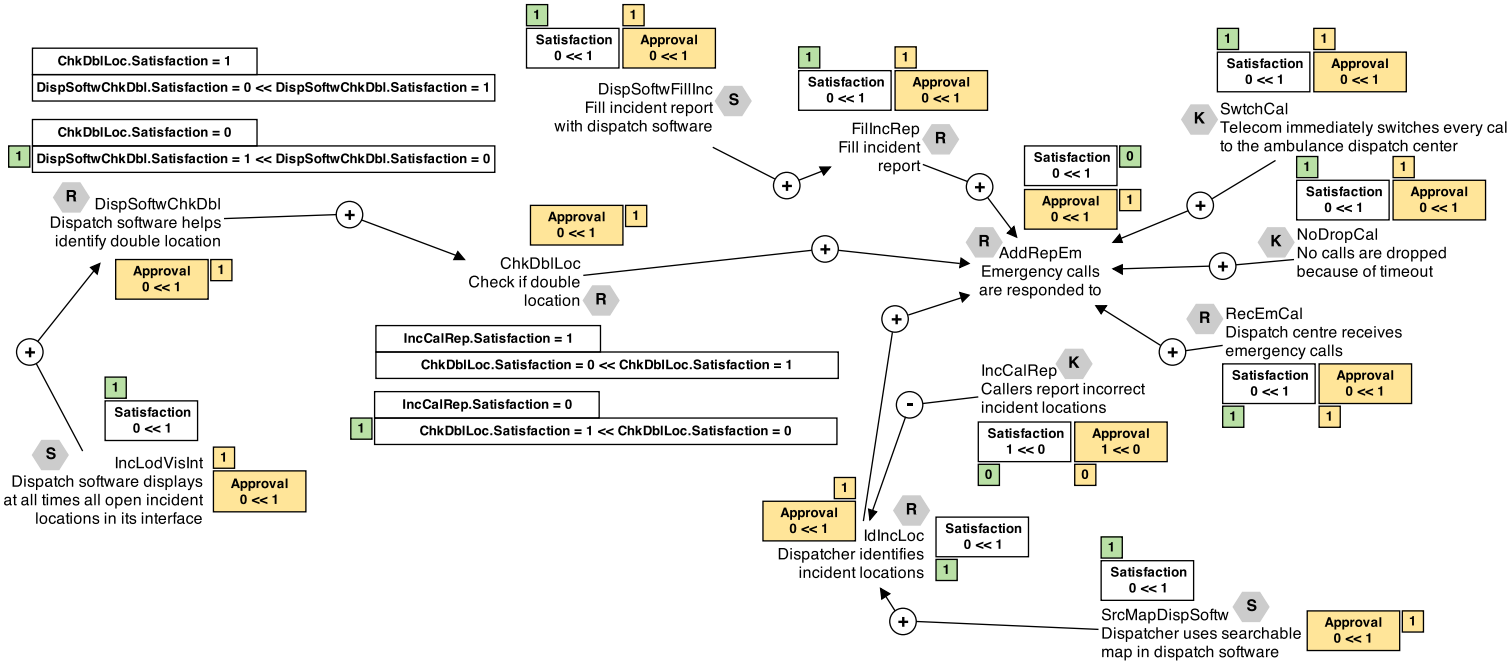


Figure 29: Best Outcome which ignores conditional preferences.

- How to represent acceptable tradeoffs between preferences, which are the allowed improvements on one preference, and the acceptable decreases over others, which are in conflict with the first?
- How to use tradeoffs when searching for the best Outcome?

12 Formal Theories

Overview and Motivation

This section is on how to relate languages in this tutorial to formal logics. The convention below is that a formal theory, or simply theory, is a name for a set of formulas with no free variables, in some formal logic. So how can you map (parts of) models to theories, and why do so?

Relationships between RMLs and formal logics are a recurrent topic in RE. In *KAOS*, theories in linear temporal first-order logic are themselves parts of models. Same in *Tropos*. The motivation is that you can take a model in an RML, map (parts of) it to a theory in some formal logic, in order to answer questions which your RML could not.

I will look at two among many topics on the relationships between RMLs and formal logics. I restrict the discussion to one formal logic, namely classical propositional logic (CPL) and discuss the following questions.

1. How to map a model to a CPL theory, if every Fragment equates to an atomic proposition? (Section 12.1),

2. How to map a model to CPL theory, if every Fragment maps to a conjunction of formulas of classical propositional logic? (Section 12.2),

3. What can be the risks of mapping models to theories? (Section 12.3).

The overall aim of mapping models to theories is to deliver Language Services which the RML could not deliver by itself. For example, I had no notion of consistency or inconsistency in languages which I introduced so far. To check if a model (part) is consistent, I need to have a notion of consistency in the language, that is, define the conditions that a model has to satisfy, in order to be consistent. I can do this independently of any existing notion of consistency in another language, or a formal logic. Or, I can map my models to theories of a formal logic, and consider my models as consistent in my language, if the corresponding theories are consistent in the formal logic. That is, I borrow a notion of consistency from an existing language or logic.

To be more concrete, recall that many languages defined so far can distinguish between Fragments that are requirements, domain knowledge, or specifications. To represent instances of the DRP, a language also needs to be able to check if requirements, domain knowledge, and the specification are consistent, and there is a proof of all requirements from the domain knowledge and the specification. This is summarised in the following Language Service.

Language Service

DRPSol: Given a model M which includes an instance P of DRP, is the part S of that model a solution to the problem instance?

s.DRPSol requires two capabilities, one related to proving requirements from domain knowledge and specifications, the other proving the absence of inconsistency. To avoid confusion about these, *Provability Condition* abbreviates hereafter the first condition in the DRP, and *Consistency Condition* the second condition.

To enable a language to deliver s.DRPSol, you need to define rules for constructing proofs, and in relation to these rules, defining when inconsistency is the result of a proof.

A cautious approach to delivering s.DRPSol is to map the content of a Requirements Model to formulae in a formal logic, where the notions of proof and inconsistency already are well-defined. The clear benefit is that you are freed from the burden of inventing a new set of proof rules and justifying them. The risk is that you may be adopting the conventions of the formal logic, and they may be clashing with the conventions in the language you use. I return to this issue in Section 12.2.

The cautious approach has the effect that you do not need to add *new* relations to models. In other words, you will still be saying the same with your models, and you will use logic *only* to deliver s.DRPSol. The other way could have involved adding new relations to the language *because* of the ability of the formal logic to state such relations. In brief, I focus on mapping models to formulae of logic, not the other way round. For the sake of simplicity, I will be mapping models to CPL theories [112].[9]

[9] Since I want to have a language that represents instances of the DRP, and not some other class of RPs, a disclaimer is in order: The syntactic consequence relation in classical propositional logic is usually denoted $\vdash$, and this at least visually resembles the relation in DRP. I do not know exactly *which* logic the DRP takes that relation from, as the accompanying paper [140] does not say. I take classical propositional logic to be a conservative choice.

12.1 How to Map Models to Theories When Fragments Map to Atomic Propositions?

Suppose that I have a model in a language which can represent Fragments and positive and negative influence relations over Fragments. Since I am interested in the DRP, the language should also categorise all Fragments into requirements, domain knowledge, and specifications. A simple language which has this is L.Rigel. Recall that L.Rigel also has v.Satisfaction and functions which propagate these values over relation instances and Fragments.

Exercise 30: Map L.Rigel models to propositional logic theories

What do you need to add to L.Rigel in order to map its models to propositional logic theories? Can the same model be mapped to different theories? If yes, then why would you map it to one of these theories and not another?

I start with the convention that the formal theory should have exactly one atomic proposition per Fragment in a model. So a Fragment is not rewritten into a sentence of CPL, but maps to an atomic proposition of CPL.

I will map positive influences to an implication from a conjunction, and negative influences to implication to inconsistency. This is inspired by *Techne*. The following function does it.

Function

Map positive influences to implications, negative influences to inconsistency
(f.map.infl.impl)

Input

Model M.

Do

1. Let Δ be an empty set.
2. For every Fragment x in M:
   (a) Let $\{(p_1,x),\ldots,(p_n,x)\}$ be the set of all positive influences to x in M.
   (b) Let $\{(q_1,x),\ldots,(q_m,x)\}$ be the set of all negative influences to x in M.
   (c) Add the following sentences to Δ:

$$p_1 \wedge \ldots \wedge p_n \rightarrow x,$$
$$q_1 \wedge \ldots \wedge q_m \wedge x \rightarrow \bot.$$

Output

Set Δ of propositional logic sentences.

Language Services

- **s.WhCPLTh**: What CPL theory corresponds to positive and negative influences over Fragments in M? : Δ.

f.map.infl.impl sees positive influences as implications, because that roughly corresponds to the idea that if $p_1,\ldots,p_n$ are satisfied, then so should x. In contrast, it sees negative influences as logical inconsistencies, so that if $q_1,\ldots,q_m$ negatively influence x, then there can be no consistent model which includes all of them. Negative influences, in this approach, should not be tolerated in solutions.

If you add f.map.impl.infl to L.Rigel, you have a language which can deliver s.DRPSol. It is called L.Sirius and defined below. Delivering it involves finding in a model an instance of the Default RP, and then being able to check if a sub model is a solution, that is, satisfies the Provability Condition and the Consistency Condition.

The convention is that a model M' in some language is a submodel of a model M in the same, or another language, if M' can be obtained by only removing Fragments and, or relation instances from M.

Language

Sirius

Language Modules

F, **T**, **V**, r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, f.inf.sat.pos, f.inf.sat.neg, f.sat, f.sat.leaf, f.map.infl.impl

Domain

Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathsf{F} = \mathsf{c.r} \cup \mathsf{c.k} \cup \mathsf{c.s}$ and $\mathsf{c.r} \cap \mathsf{c.k} \cap \mathsf{c.s} = \emptyset$. Influences are over Fragments, $\mathsf{r.inf.pos} \subseteq \mathsf{F} \times \mathsf{F}$, $\mathsf{r.inf.neg} \subseteq \mathsf{F} \times \mathsf{F}$. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\mathbf{V} \subseteq (\mathsf{F} \cup \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg}) \times \mathbf{T} \times \mathsf{v.Satisfaction}.$$

The language has one binary Value Type, $\mathbf{T} = \{\mathsf{v.Satisfaction}\}$, and $\mathsf{v.Satisfaction} = \{1, 0\}$.

Syntax

A model M in the language is a set of symbols $M = \{\phi_1, \ldots, \phi_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} \alpha & ::= & x \mid y \mid z \mid \ldots \\ \beta & ::= & r \mid k \mid s \\ \gamma & ::= & \beta(\alpha) \\ \delta & ::= & (\gamma, \gamma) \\ \epsilon & ::= & \langle \alpha, \zeta, \eta \rangle \\ \phi & ::= & \gamma \mid \delta \mid \epsilon \end{array}$$

Mapping

α symbols denote Fragments, $\mathscr{D}(\alpha) \in \mathsf{F}$. β symbols are used to distinguish requirements, domain knowledge, and specification Fragments, and are used in γ symbols, so that $\mathscr{D}(r(\alpha)) \in \mathsf{c.r}$, $\mathscr{D}(k(\alpha)) \in \mathsf{c.k}$, $\mathscr{D}(s(\alpha)) \in \mathsf{c.s}$. δ symbols denote positive and negative influence relations. ζ symbols denote Value Types, $\mathscr{D}(\zeta) \in \mathbf{T}$. η denotes a value of a Value Type, and as there is one Value Type, then $\mathscr{D}(\eta) \in \mathsf{v.Satisfaction}$. ϵ symbols denote value assignments, $\mathscr{D}(\epsilon) \in \mathbf{V}$.

Language Services

Those of relations and functions in the language, and

- s.DRPSol: Yes, S is the solution to the Default RP instance defined by the sub model P, if the following conditions are satisfied:
  1. P and S are submodels of M,
  2. If $\mathsf{F_{c.k}}$ is the set of atomic CPL propositons, one per domain knowledge Fragment in M, $\mathsf{F_{c.s}}$ the set of atomic CPL propositons, one per specification Fragment in M, $\mathsf{F_{c.r}}$ the set of atomic CPL propositons, one per requirement Fragment in M, and Δ the set of CPL sentences produced by applying f.map.imfl.impl to M, then
     (a) $\mathsf{F_{c.k}}, \mathsf{F_{c.s}}, \Delta \vdash \mathsf{F_{c.r}}$, that is, the Provability Condition is satisfied,
     (b) $\mathsf{F_{c.k}}, \mathsf{F_{c.s}}, \Delta \nvdash \bot$, that is, the Consistency Condition is satisfied,
  3. S includes all Fragments denoted by the atomic propositions in $\mathsf{F_{c.s}}$, and
  4. P includes all Fragments denoted by the atomic propositions in $\mathsf{F_{c.k}} \cup \mathsf{F_{c.r}}$.

Finding an Default RP instance in a model in L.Rigel is simple. If the model has a set of requirements Fragments and domain knowledge Fragments, then it includes a Default RP instance. There was no need for f.map.impl.infl to do this.

Recall that the Provability Condition consists of showing that $K, S \vdash R$. Let K be the set of all domain knowledge Fragments in M, S of specification Fragments, and R of requirements. You then need to have an atomic proposition for each Fragment, so let $\mathsf{F_{c.k}}$ be the set of atomic CPL propositons, one per domain knowledge Fragment in M, $\mathsf{F_{c.s}}$ the set of atomic CPL propositons, one per specification Fragment in M, $\mathsf{F_{c.r}}$ the set of atomic CPL propositons, one per requirement Fragment in M.

None of these sets includes influence relations, and it follows, cannot include Δ, the implications which correspond to the positive and negative influences in M. The revised Provability Condition is then to show that

$$\mathsf{F_{c.k}}, \mathsf{F_{c.s}}, \Delta \vdash \mathsf{F_{c.r}}.$$

The Consistency Condition becomes

$$\mathsf{F_{c.k}}, \mathsf{F_{c.s}}, \Delta \nvdash \bot.$$

In a summary, if M gives the sets R, K, and S of propositions, and via f.map.infl.impl the set of implications Δ, and if it can be shown that the above two conditions are satisfied, then S is the solution to the Default RP in M.

Note that M may include other Fragments and relations, but if only f.map.infl.impl is used, then M will be logically inconsistent if $F, \Delta \vdash \bot$, where F are all the Fragments in M. It follows that M may be inconsistent, all the while $K, S, \Delta \vdash R$ and $K, S, \Delta \nvdash \bot$.

Exercise 31: Given formulas, find the corresponding L.Sirius model

Suppose that you are given the following CPL formulas:

$$
\begin{aligned}
\mathsf{F_{c.k}} &= \{ \quad \mathsf{SwtchCal}, \mathsf{NoDropCal} \ \} \\
\mathsf{F_{c.s}} &= \{ \quad \mathsf{SrcMap}, \mathsf{IncLocVisSw}, \mathsf{UpdOpIncLoc}, \mathsf{FilSwIncRep} \ \} \\
\mathsf{F_{c.r}} &= \{ \quad \mathsf{AddRepEm} \ \} \\
\Delta &= \{ \quad \mathsf{FilIncRep} \wedge \mathsf{ChkDblLoc} \wedge \mathsf{IdIncLoc} \wedge \mathsf{RecEmCal} \wedge \mathsf{NoDropCal} \wedge \mathsf{SwtchCal} \\
&\qquad \rightarrow \mathsf{AddRepEm}, \\
&\qquad \mathsf{SwtchCal} \wedge \mathsf{NoDropCal} \rightarrow \mathsf{RecEmCal}, \\
&\qquad \mathsf{SrcMap} \rightarrow \mathsf{IdIncLoc}, \\
&\qquad \mathsf{UpdOpIncLoc} \wedge \mathsf{IncLocVisSw} \rightarrow \mathsf{SwIdDuplCal}, \\
&\qquad \mathsf{SwIdDuplCal} \rightarrow \mathsf{ChkDblLoc}, \\
&\qquad \mathsf{FillSwIncRep} \rightarrow \mathsf{FillIncRep} \ \}.
\end{aligned}
$$

What Fragments and relations are in a model M, if it is a model in L.Sirius, and which includes only the Fragments that correspond to atomic propositions in $\mathsf{F_{c.k}}$, $\mathsf{F_{c.s}}$, and $\mathsf{F_{c.r}}$ above, and has influences which mapped to those in Δ above, via f.map.inf.impl? Is there a Default RP problem instance in M? Is S given above a solution to that Default RP instance in M?

12.2 What If Fragments Map to Sentences?

Instead of mapping each Fragment an atomic proposition, what would happen if you mapped a Fragment to a CPL sentence?

Suppose, then, that there is a function which takes a Fragment and returns a sentence. Call it f.map.f.sntc. I have no suggestions on how to define this function, other than that the modeller takes a Fragment and writes a CPL sentence which best corresponds to the information in the Fragment.

The effect of having f.map.f.sntc is that R, K, and S are now sets of sentences. You still need to map relations to sentences, and f.map.inf.impl can still be used, with the change that implications are now not necessarily only over atomic propositions, but over atomic propositions and, or sentences.

Changes to the problem are the same as in Section 12.1. Provability Condition is $K, S, \Delta \vdash R$ and the Consistency Condition is $K, S, \Delta \nvdash \bot$.

This has an effect on the complexity of checking the two conditions. The check could be done in linear time when the output are atomic propositions and implications in Section 12.1, since that output amounts to a set of propositional Horn clauses [40].

12.3 Are There Risks of Mapping Models to Theories?

Suppose that you have L.Rigel and f.map.inf.impl, and that you map Fragments to atomic propositions of CPL, as in Section 12.1.

Let x be a Fragment, and an atomic proposition which is a requirement in a model M. You might want to check if

$$K, \Delta \vdash x,$$

and if yes, conclude that the requirement x is satisfied by the domain knowledge. More generally, you may want to check if $M' \vdash x$, where M' is the mapping of the model M to atomic propositions and implications.

There is nothing problematic with wanting to do this, but it can be misleading. The syntactic consequence relation $\vdash$ in CPL is reflexive, meaning that if $x \in M'$, then also $M' \vdash x$. So even if there are no implications from domain knowledge and specifications to x, and thus, no clear idea how to satisfy the requirement x with M, it is the case that $M' \vdash x$. The danger is to conclude that x is a satisfied requirement according to M' and therefore, that M says how to satisfy x. This is incorrect.

A similarly misleading case is if $M' \vdash \bot$. When M' is inconsistent, then any atomic proposition and sentence is its conclusion in CPL. So any y, be it in M or not, is such that $M' \vdash y$. If you were reading $M' \vdash x$ as indicating that M says how to satisfy the requirement x, then you would conclude anytime you have an inconsistent M', regardless of there being x in it, or not, or there being domain knowledge and specifications in M which say how to satisfy x.

The odd cases above happen not because there is a problem with the formal logic, or with the RML, but with the rules about how the two are related. For example, if M' can be inconsistent, then it might be interesting to use a paraconsistent logic rather than CPL, to check if there is proof of x from M'. In short, the choice of a formal logic to map models to, depends on exactly what you want to use this logic for, which consequently helps choose that formal logic.

12.4 Summary on Formal Theories

This section briefly mentioned several topics on how models in RMLs relate to theories in formal logics. The central idea and motivation is that (parts of) models can be mapped to theories of formal logic. It should then be possible to check properties of these theories, such as consistency, to draw conclusions which help change the original models. I leave many other questions outside the scope of this tutorial:

- How does valuation in a language influence the choice of a formal logic to map its models to?
- Which properties of a language influence one's decision on what to map Fragments and relations to, in a formal logic?
- When models can map to inconsistent theories, then which paraconsistent logic to map the models to, in order to do reasoning without repairing consistency first?

13 Problem Classes

Overview and Motivation

This section presents various RP classes, some similar, others different from the Default RP. The aim is to illustrate that the design of problem classes and the design of languages is not independent from each other. Languages can be designed to solve specific problem classes. Problem classes can also be defined after a language is, to fit and showcase the capabilities of that language. This can happen, for example, if you designed the language without a specific problem class in mind, but perhaps a set of Language Services which you are interested in. These Language Services may not, by themselves, define a problem class. This is exactly how I designed all languages in this tutorial, except for those made specifically to solve the Default RP, in Section 12. To clarify these ideas, I discuss the following specific questions.

1. What is a RP class, and why and how to define one? (Section 13.1),
2. How languages and problem classes match? (Section 13.2),
3. How to match problem classes and solution procedures? (Section **??**),
4. What and how can problem classes inherit from each other? (Section 13.3).

In all languages so far in this tutorial, Language Services were only indirectly related to the specific RPs that I may wanto to solve. For example, I motivated the use of categories by requiring a language to distinguish requirements, domain knowledge, and specifications, because it cannot otherwise be used to solve Default RP instances. But the ability to distinguish categories is only a small part of what a language should be capable of, in order to identify and solve Default RP instances in models. For example, it has to be able to identify classically inconsistent sets of formulas, which has nothing to do with the ability to categorise Fragments (as I illustrated in Section 12). So the Language Services which ensured that these categories are distinguished in models are only indirectly helping, that is, are only part of what is needed to solve Default RP instances.

Take another example. If I have the Language Service that consists of finding all acceptable Fragments in a model, then it alone tells me nothing about whether an acceptable Fragment should be in a solution to the problem, what that problem may be in the model I have, and what the solution may be. If a language delivers the Language Service which gives all acceptable Fragments in a model, then this alone is not enough to know what the problem and the solution are. Is the problem to find all acceptable Fragments in a model? Is it to find all acceptable Fragments which are categorised as requirements? Or is it to check if the set of all acceptable Fragments includes all those in some special relation to all requirements Fragments in a model?

It is important to know if a model represents an instance of a problem class, and if the model includes the solution to that problem instance. If you know the problem class that the language can represent and solve, then you may want to elicit the Fragments, and establish relations, which together represent a problem instance. Once you have a problem instance, you may want to elicit or otherwise find information about potential solutions, that is, the alternative specifications that the model may include. The solution will be the one specification you choose to commit to.

If the language has Language Services, then knowing the problem helps decide which Language Services to mobilise when, in order to represent the problem instance and search for its solution. For example, if the problem is to make sure that all requirements Fragments in a model obtain the value “satisfied”, then you would need first to find and categorise as requirements some of the Fragments in the model. Then, at various times during problem solving, you would need to check which of the requirements Fragments obtain the desired value. If some do not, then you have not found the solution yet, and you need to focus next problem solving steps on changing the model, so that it satisfies more of the requirements Fragments. It is also useful to know the problem class before you define the Language Services. If the problem class is Default RP, then finding Problems requires mobilising Language Services which categorise Fragments into requirements, domain knowledge, and specifications, Language Services which check if a set of formulas is consistent, and so on. Knowing the problem class helps focus the search for relevant Language Services that a language should deliver.

In a summary, it is relevant when designing a language to indicate which one or more problem classes it can deal with. This helps make sense of the various Language Services that the language may deliver, in that they all, or most need to be useful towards identifying the problem in a model, and finding its solutions.

13.1 How to Define Requirements Problem Classes?

The definition of an RP class has two parts:

1. a set of conditions which a model needs to satisfy, in order to represent an instance of a problem of that problem class, and

2. a set of conditions which a model needs to satisfy, in order to include the solution to the problem instance of that problem class.

In the rest of this tutorial, the term Problem will denote an instance of a problem, of some problem class, and Solution will denote an instance of the solution to the problem of a problem class. So given some model M in some language, a Problem is some part of M which satisfies the conditions of being a problem instance, of some problem class, and Solution is some part of M which satisfies the conditions of being an instance of the solution, of a problem class.

There are two obvious and alternative ways to say that a language can represent Problems and Solutions of a problem class. One is to add two functions. You give each a submodel. One checks if a submodel is a Problem. The other checks if it is a Solution.

I prefer another way, which is to have special Language Modules for problem classes, rather than reuse those for functions. The aim is simply to emphasise problem classes by not calling them functions. So if you have a problem class C, then its module needs to deliver the following Language Services:

- **s.IsProblem**: Does the model M include the Problem P of the problem class C?
- **s.IsSolution**: If the model M includes the Problem P of problem class C, does M also include the Solution S to this Problem?

Take Default RP as an example. A model includes a Default RP Problem if it includes requirements and domain knowledge Fragments. The model also includes a Default RP Solution to that problem, if it includes specification Fragments, and if it can be shown with that language that the Provability Condition and the Consistency Condition are satisfied.

Suppose now that you have L.Rigel, where there is no syntactic consequence relation and no rules to map its models to a formal theory (in contrast to Section 12). The Provability Condition and the Consistency Condition cannot be verified for L.Rigel models and the language cannot represent Default RP Problems and Solutions. But I can define a problem class which is inspired by the consistency and satisfaction conditions, and can be represented withL.Rigel. The language distinguishes requirements, domain knowledge, and specifications, so I can define the Problem for the problem class just as in Default RP.

In that language, the Problem is such a part of a model, which includes all requirements and domain knowledge, and all relations that only those Fragments are in.

The Solution should satisfy two conditions inspired by the Default RP. Call them "light satisfaction" and "light consistency".

Light satisfaction is satisfied if there is a value assignment to leaf Fragments which propagate the v.Satisfaction 1 to all requirements Fragments, and these leaf Fragments are specifications and, or domain knowledge.

Light consistency is satisfied if there are no negative influences between specifications, requirements, and domain knowledge. This also means that, if there are negative influences in the Problem, the problem has to be revised as well, and you cannot find the solution simply by adding specifications and positive influence relations. The following Language Module synthesises this.

Problem Class

DRPLight (pcl.DRPLight)

Input

Three models M, P, and S.

Problem

Model M defines the Problem P of class pcl.DRPlight if the following conditions are satisfied.

1. P is a submodel of M.
2. All c.r and c.k Fragments in M are also in P.
3. All relation instances over c.r and c.k Fragments only (and not over other categories of Fragments) in M are also in P.

If P satisfies the conditions above, then $v = 1$, else $v = 0$.

Solution

Model M defines the Solution S to the Problem P of pcl.DRPlight if the following conditions are satisfied.

1. S is a submodel of M.
2. Of all Fragments in M, S includes only c.s Fragments.
3. Of all relation instances in M, S includes only those over only c.s Fragments.
4. *Light satisfaction:* There is a value assignment to leaf Fragments in S which propagate the v.Satisfaction 1 to all requirements Fragments in P,

and these leaf Fragments in S are specifications and, or domain knowledge.

5. *Light consistency:* There are no negative influences between specifications, requirements, and domain knowledge in the sub model of M, which includes only the Fragments in S and P and all relations over these Fragments.

If S satisfies the conditions above, then $w = 1$, else $w = 0$.

Language Services

- **s.IsProblem.DRPlight**: Does the model M include the Problem P of pcl.DRPlight? Yes, if $v = 1$, no otherwise.
- **s.IsSolution.DRPlight**: Does the model M include the Solution S of the Problem P of pcl.DRPlight? Yes, if $v = 1$ and $w = 1$, no otherwise.

If the language was L.Sirius, and the Fragments mapped to atomic propositions, then you could define the Default RP as follows.

Problem Class

DRP (pcl.DRP)

Input

Three models, M, P, and S.

Problem

Model M defines the Problem P of class pcl.DRP if P only all requirements and domain knowledge Fragments of M. That is, $P = R \cup K$, where K is the set of all domain knowledge Fragments in M, and R the set of all requirements Fragments in M.

If P satisfies the condition above, then $v = 1$, else $v = 0$.

Solution

Model M defines the Solution S of class pcl.DRP, if S satisfies the following conditions:

1. S is a sub model of M.
2. S includes only specification Fragments and no relation instances.
3. The Provability Condition is satisfied, that is, $K, S, \Delta \vdash R$, where M maps to the set Δ of CPL sentences via f.map.inf.impl.
4. The Consistency Condition is satisfied, that is, $K, S, \Delta \nvdash \bot$.

If S satisfies the conditions above, then $w = 1$, else $w = 0$.

Language Services

- **s.IsProblem.DRP**: Does the model M include the Problem P of pcl.DRP? Yes, if $v = 1$, no otherwise.
- **s.IsSolution.DRP**: Does the model M include the Solution S of the Problem P of pcl.DRP? Yes, if $v = 1$ and $w = 1$, no otherwise.

13.2 Why Match Problem Classes and Languages?

A problem class is not necessarily specific to one language. Models of a language may include Problems of different problem classes. It can also happen that models include Problems of a problem class, yet the language is unable to represent Solutions of that problem class. I will illustrate these ideas below. The aim is to show that it is important to take care when deciding how generic (language-independent) a problem class is, and how versatile a language can be claimed to be, that is, which problem classes it can be used to solve.

As a first example, observe that any language, which can represent requirements and domain knowledge Fragments, can also represent Problems of the pcl.DRPlight problem class. This means, for instance, that a language which categorises Fragments into requirements, domain knowledge, and specifications, and only allows positive and negative r.arg relations over Fragments, can represent pcl.DRPlight Problems. But it cannot represent Solutions, because it has no relations which can indicate that satisfying specifications satisfies requirements and, or domain knowledge, and no functions to propagate satisfaction values.

Any language that can represent requirements and domain knowledge Fragments can also represent Default RP Problems. But if the language has a syntactic conse-

quence relation which is different from that of classical logic (say, it has an irreflexive consequence relation, or a paraconsistent one), then it can also have a different notion of consistency, which makes it impossible to check if a submodel includes a Default RP Solution.

Suppose, for example, that $\vdash_A$ is a consequence relation in one of Hunter's argumentative logics [70], and $\vdash$ is the consequence relation of CPL (as in Section 12). If you require a Solution to satisfy $K, S \vdash_A R$, then, clearly, you are asking it to satisfy a different condition than $K, S \vdash R$, because the rules that K, S, and R have to satisfy in the former case are different than those that they have to satisfy in the latter case. Simply put, the two syntactic consequence relations are different, and so, the properties of Solutions are different, and finally, you have different problem classes.

As a second example, note that pcl.DRPlight requires that all requirements and all domain knowledge Fragments are in every Solution. So you cannot choose a subset thereof, and therefore, you prefer equally all requirements and domain knowledge.

pcl.DRPlight is not specific to L.Rigel, yet it puts in the Problem any relations over requirements and domain knowledge. This is fine when there are only influence relations in a language. But what if there were also r.xor instances? If there were, then there could be alternative subsets of requirements and domain knowledge, and each one is a different Problem. This is simply because there is no need, or it is impossible for a Solution to satisfy all alternative requirements. The language might also have acceptability relations and a function to compute the acceptability value. Perhaps it makes sense that the Problem includes only the acceptable requirements and domain knowledge. Secondly, light consistency is not light at all. It requires that there be no negative influences over requirements, domain knowledge, and specifications. This may be too much to ask, and if so, a different Solution concept is needed. The module for pcl.DRPlight has no rules for how to deal with cases, where some negative influences are allowed in a Solution.

It can also happen that a model can represent several mutually exclusive Solutions to the same Problem. For example, a model in L.Capella can represent Alternatives and Combinations, and propagate satisfaction values over them. More importantly, its models can represent Problems of pcl.DRPlight, since they can show requirements Fragments. There may be such value assignments, which guarantee that different Combinations will, each, satisfy all the requirements, and so, each Combination may itself include a Solution to that pcl.DRPlight Problem. The definition of pcl.DRPlight does not say which of these Solutions is *the* Solution. That is, any of them is. If the model includes preferences, then some of these Solutions, that is, Outcomes, may be preferred to others. However, pcl.DRPlight ignores preferences, and therefore, a different problem class may be more appropriate.

The general point above is that problem classes should ideally use the features of the language, so that the intention to use the language to solve problems of a given class, is a motive for having these features in the first place. Below is the problem class pcl.DRPbest, which adapts pcl.DRPlight so that it uses preferences, that is, the solution has also to give the most preferred Outcome.

Problem Class

DRPbest (pcl.DRPbest)

Input

Three models, M, P, and S.

Problem

Model M defines the Problem P of class pcl.DRPbest if M defines the problem P of pcl.DRPlight.

If P satisfies the condition above, then $v = 1$, else $v = 0$.

Solution

Model M defines the Solution $O(M, S)$ of class pcl.DRP, if the following two conditions are satisfied.

1. S is a Solution to P according to pcl.DRPlight.
2. Let $O(M, S)$ and $O(M, S')$ be two satisfaction Outcomes on M. That is, each includes a single satisfaction value assignment to every Fragment and relation instance in M. Let S and S' be sub models of M, such that each is a Solution to P according to pcl.DRPlight. So if you were looking for a pcl.DRPlight Solution to P, you could take either S or S'. There is at no preference relation in M such that $O(M, S')$ is strictly preferred to $O(M, S)$.

If $O(M, S)$ satisfies the conditions above, then $w = 1$, else $w = 0$.

Language Services

- **s.IsProblem.DRPbest**: Does the model M include the Problem P of pcl.DRP? Yes, if $v = 1$, no otherwise.

- **s.IsSolution.DRPbest**: Does the model M include the Solution $O(M,S)$ of the Problem P of pcl.DRP? Yes, if $v = 1$ and $w = 1$, no otherwise.

13.3 What and How Can Problem Classes Inherit from Each Other?

Work in progress. Will be added soon.

14 Discussion

Work in progress. Will be added soon.

A Language Modules and Languages in the Tutorial

Tables 3 and 4 show all languages and Language Modules defined in this tutorial.

Table 3: All Language Modules and languages in this tutorial.

Language Module	Section	Language																		
		L.Alpheratz	L.Ankaa	L.Schedar	L.Diphda	L.Achernar	L.Hamal	L.Acamar	L.Menkar	L.Mirfak	L.Aldebaran	L.Rigel	L.Capella	L.Adhara	L.Bellatrix	L.Elnath	L.Alnilam	L.Canopus	L.Procyon	L.Sirius
Relations	5																			
r.ifm	5.1	•			•		•	•												
f.map.abrel.g	5.2			•	•	•			•	•	•	•	•	•	•	•	•	•	•	•
r.inf	5.3.1																			
r.inf.pos	5.3.2		•	•		•		•	•	•	•	•	•	•	•	•	•	•	•	•
r.inf.neg	5.3.2		•	•		•		•	•	•	•	•	•	•	•	•	•	•	•	•
r.str.inf	5.3.3			•					•											
r.sup	5.4.1				•															
r.def	5.4.1				•															
f.acc	5.4.2																			
Guidelines	6																			
f.add.ifm	6.1						•	•												
r.q	6.1						•													
f.opr.all	6.2							•												
Categories	7																			
c.r	7.1								•	•	•	•	•	•	•	•	•	•	•	•
c.k	7.1								•	•	•	•	•	•	•	•	•	•	•	•
c.s	7.1								•	•	•	•	•	•	•	•	•	•	•	•
f.cat.ksr	7.1								•	•	•	•	•	•	•	•	•	•	•	•
c.r.f	7.2																			
c.r.nf	7.2																			
r.rls	7.4																			
c.r.clsh	7.4																			
c.r.irrl	7.4																			
f.chk.rop	7.5																			
Alternatives	8																			
r.xor	8.1									•	•						•			
r.po	8.2										•									
c.cp	8.2										•									
c.cb	8.3									•	•									

Table 4: All Language Modules and languages in this tutorial (continued).

Language Module	Section	Language																		
		L.Alpheratz	L.Ankaa	L.Schedar	L.Diphda	L.Achernar	L.Hamal	L.Acamar	L.Menkar	L.Mirfak	L.Aldebaran	L.Rigel	L.Capella	L.Adhara	L.Bellatrix	L.Elnath	L.Alnilam	L.Canopus	L.Procyon	L.Sirius
Valuation	9																			
f.inf.sat.pos	9.1.2											•	•	•	•	•	•	•	•	•
f.inf.sat.neg	9.1.2											•	•	•	•	•	•	•	•	•
f.sat	9.1.3											•	•	•	•	•		•	•	•
f.sat.leaf	9.1.3											•		•	•	•	•	•	•	•
f.sat.x	9.1.4												•				•			
f.app.asg.ind	9.2.1																			
f.app.maj	9.2.1																			
f.sat.nxt	9.2.1																			
f.ask.next	9.3																			
f.chk.progrstatus	9.4																			
f.chk.20more	9.5																			
Uncertainty	10																			
f.prob.asg	10.1													•						
f.prob.sat.ind	10.1													•						
f.make.baynet	10.2.1																			
f.map.inf.pos.baynet	10.2.2																			
Preferences	11																			
r.pref.loc.c	11.2														•	•	•	•	•	
r.pref.mloc.c	11.3															•	•	•	•	
r.pref.br.c	11.4																•	•	•	
r.pref.mbr.c	11.5																	•	•	
crit.low.AddIncTm	11.6																			
crit.d.t	11.6																			
crit.low.rev.h	11.6																			
r.pref.cond	11.7																		•	
f.makr.CPNet	11.7																			
Formal Theories	12																			
f.map.infl.impl	12.1																			•
Problem Classes	13																			
pcl.DRPlight	13.1																			
pcl.DRP	13.1																			
pcl.DRPbest	13.2																			

References

[1] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] Anonymous. Report of the Inquiry Into The London Ambulance Service. Technical report, The Communications Directorate, South West Thames Regional Authority, 1993.

[3] Chimay Anumba, John M Kamara, and Anne-Francoise Cutting-Decelle. *Concurrent engineering in construction projects*. Routledge, 2006.

[4] Kenneth J Arrow, Amartya Sen, and Kotaro Suzumura. *Handbook of Social Choice & Welfare*, volume 2. Elsevier, 2010.

[5] Fahiem Bacchus and Adam Grove. Graphical models for preference and utility. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 3–10. Morgan Kaufmann Publishers Inc., 1995.

[6] Jorgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2002.

[7] Trevor JM Bench-Capon and Paul E Dunne. Argumentation in artificial intelligence. *Artificial intelligence*, 171(10):619–641, 2007.

[8] William L Benoit and Dale Hample. *Readings in argumentation*, volume 11. Walter de Gruyter, 1992.

[9] Patrik Berander and Anneliese Andrews. Requirements prioritization. In *Engineering and managing software requirements*, pages 69–94. Springer, 2005.

[10] Paul Beynon-Davies. Human error and information systems failure: the case of the london ambulance service computer-aided despatch system project. *Interacting with Computers*, 11(6):699–720, 1999.

[11] Barry Boehm, Prasanta Bose, Ellis Horowitz, and Ming June Lee. Software requirements negotiation and renegotiation aids: A theory-w based spiral approach. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 243–243. IEEE, 1995.

[12] Barry W Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.

[13] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[14] Barry W Boehm, John R Brown, and Myron Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[15] Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, 2000.

[16] Craig Boutilier, Ronen I Brafman, Carmel Domshlak, Holger H Hoos, and David Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res.(JAIR)*, 21:135–191, 2004.

[17] Jean-Pierre Brans and Ph Vincke. Note: A preference ranking organisation method: (the promethee method for multiple criteria decision-making). *Management science*, 31(6):647–656, 1985.

[18] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, 2009.

[19] Russel E Caflisch. Monte carlo and quasi-monte carlo methods. *Acta numerica*, 7:1–49, 1998.

[20] Colin Camerer and Martin Weber. Recent developments in modeling preferences: Uncertainty and ambiguity. *Journal of risk and uncertainty*, 5(4):325–370, 1992.

[21] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information systems*, 27(6):365–389, 2002.

[22] Eugene Charniak. Bayesian networks without tears. *AI magazine*, 12(4):50, 1991.

[23] Li Chen and Pearl Pu. Survey of preference elicitation methods. In *Ecole Politechnique Federale de Lausanne (EPFL), IC/2004/67*, 2004.

[24] Betty HC Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.

[25] Carlos Iván Chesñevar, Ana Gabriela Maguitman, and Ronald Prescott Loui. Logical models of argument. *ACM Computing Surveys (CSUR)*, 32(4):337–383, 2000.

[26] Yann Chevaleyre, Ulle Endriss, Jérôme Lang, and Nicolas Maudet. A short introduction to computational social choice. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, pages 51–69. Springer-Verlag, 2007.

[27] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[28] Jane Cleland-Huang, Raffaella Settimi, Chuan Duan, and Xuchang Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 135–144. IEEE, 2005.

[29] Jeff Conklin and Michael L Begeman. gibis: A hypertext tool for exploratory policy discussion. *ACM Transactions on Information Systems (TOIS)*, 6(4):303–331, 1988.

[30] Larissa Conradt and Christian List. Group decisions in humans and animals: a survey. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 364(1518):719–742, 2009.

[31] Richard Cox. Representation construction, externalised cognition and individual differences. *Learning and instruction*, 9(4):343–363, 1999.

[32] Oliver Creighton, Martin Ott, and Bernd Bruegge. Software cinema-video-based requirements engineering. In *Requirements Engineering, 14th IEEE International Conference*, pages 109–118. IEEE, 2006.

[33] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.

[34] Robert Darimont and Axel Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *ACM SIGSOFT Software Engineering Notes*, 21(6):179–190, 1996.

[35] Alan Davis, Oscar Dieste, Ann Hickey, Natalia Juristo, and Ana María Moreno. Effectiveness of requirements elicitation techniques: Empirical results derived from a systematic review. In *Requirements Engineering, 14th IEEE International Conference*, pages 179–188. IEEE, 2006.

[36] Victorio A de Carvalho, João Paulo A Almeida, and Giancarlo Guizzardi. Using reference domain ontologies to define the real-world semantics of domain-specific languages. In *Advanced Information Systems Engineering*, pages 488–502. Springer, 2014.

[37] Willem-Paul de Roever, Kai Engelhardt, and Karl-Heinz Buth. *Data refinement: model-oriented proof methods and their comparison*. Number 47. Cambridge University Press, 1998.

[38] Edsger W Dijkstra. Chapter i: Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press Ltd., 1972.

[39] Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in ai: An overview. *Artificial Intelligence*, 175(7):1037–1052, 2011.

[40] William F Dowling and Jean H Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[41] Didier Dubois and Henri Prade. Possibility theory as a basis for qualitative decision theory. In *IJCAI*, volume 95, pages 1924–1930, 1995.

[42] Eric Dubois, Jacques Hagelstein, Eugene Lahou, Frank Ponsaert, and Andre Rifaut. A knowledge representation language for requirements engineering. *Proceedings of the IEEE*, 74(10):1431–1444, 1986.

[43] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and< i> n</i>-person games. *Artificial intelligence*, 77(2):321–357, 1995.

[44] Neil A Ernst, Alexander Borgida, Ivan J Jureta, and John Mylopoulos. Agile requirements engineering via paraconsistent reasoning. *Information Systems*, 2013.

[45] José Figueira, Salvatore Greco, and Matthias Ehrgott. *Multiple criteria decision analysis: state of the art surveys*, volume 78. Springer, 2005.

[46] Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *Software Engineering, IEEE Transactions on*, 20(8):569–578, 1994.

[47] Janos C Fodor and MR Roubens. *Fuzzy preference modelling and multicriteria decision support*, volume 14. Springer, 1994.

[48] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Pistore, Marco Roveri, and Paolo Traverso. Specifying and analyzing early requirements in tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[49] Dedre Gentner and Susan Goldin-Meadow. *Language in mind: Advances in the study of language and thought*. MIT Press, 2003.

[50] Jonathan Ginzburg. Interrogatives: Questions, facts and dialogue. *The handbook of contemporary semantic theory. Blackwell, Oxford*, 1996.

[51] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. In *Conceptual ModelingâĂŤER 2002*, pages 167–181. Springer, 2003.

[52] Lila Gleitman and Anna Papafragou. Language and thought. *Cambridge handbook of thinking and reasoning*, pages 633–661, 2005.

[53] Joseph A Goguen and Charlotte Linde. Techniques for requirements elicitation. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 152–164. IEEE, 1993.

[54] Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101. IEEE, 1994.

[55] Salvatore Greco, Benedetto Matarazzo, and Roman Slowinski. Rough sets theory for multicriteria decision analysis. *European journal of operational research*, 129(1):1–47, 2001.

[56] Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: Rml revisited. In *Proceedings of the 16th international conference on Software engineering*, pages 135–147. IEEE Computer Society Press, 1994.

[57] Sol J Greenspan, John Mylopoulos, and Alex Borgida. Capturing more world knowledge in the requirements specification. In *Proceedings of the 6th international conference on Software engineering*, pages 225–234. IEEE Computer Society Press, 1982.

[58] Nicola Guarino. Formal ontology, conceptual analysis and knowledge representation. *International journal of human-computer studies*, 43(5):625–640, 1995.

[59] John J Gumperz and Stephen C Levinson. *Rethinking linguistic relativity*. Cambridge University Press, 1996.

[60] Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. In *Requirements engineering, 2000. Proceedings. 4th International Conference on*, page 189. IEEE, 2000.

[61] Sven Ove Hansson. Preference logic. In *Handbook of philosophical logic*, pages 319–393. Springer, 2002.

[62] Sven Ove Hansson and Till Grüne-Yanoff. Preferences. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2012 edition, 2012.

[63] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of" semantics"? *Computer*, 37(10):64–72, 2004.

[64] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.

[65] Andrea Herrmann and Maya Daneva. Requirements prioritization based on benefit and cost prediction: An agenda for future research. In *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 125–134. IEEE, 2008.

[66] Ann M Hickey and Alan M Davis. A unified model of requirements elicitation. *Journal of Management Information Systems*, 20(4):65–84, 2004.

[67] David Hitchcock. Informal logic and the concept of argument. *Philosophy of logic*, 5:101–129, 2006.

[68] C Hoare. Proof of correctness of data representations. *Language Hierarchies and Interfaces*, pages 183–193, 1976.

[69] John E Hopcroft and Robert E Tarjan. Efficient algorithms for graph manipulation. 1971.

[70] Anthony Hunter. Paraconsistent logics. In *Reasoning with Actual and Potential Contradictions*, pages 11–36. Springer, 1998.

[71] Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(4):335–367, 1998.

[72] United Kingdom Hydrograph, United Kingdom Hydrographic Office, and U S Naval Observatory. *2010 Nautical Almanac: Commercial Edition*. Paradise Cay Publications, 2009.

[73] Kenneth E Iverson. Notation as a tool of thought. *ACM SIGAPL APL Quote Quad*, 35(1-2):2–31, 2007.

[74] David Jonassen. Using cognitive tools to represent problems. *Journal of research on Technology in Education*, 35(3):362–381, 2003.

[75] Ivan Jureta, John Mylopoulos, and Stéphane Faulkner. Analysis of multi-party agreement in requirements validation. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 57–66. IEEE, 2009.

[76] Ivan J Jureta, Alexander Borgida, Neil A Ernst, and John Mylopoulos. Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 115–124. IEEE, 2010.

[77] Ivan J Jureta and Stéphane Faulkner. Clarifying goal models. In *Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling-Volume 83*, pages 139–144. Australian Computer Society, Inc., 2007.

[78] Ivan J Jureta, Stéphane Faulkner, and Pierre-Yves Schobbens. Clear justification of modeling decisions for goal-oriented requirements engineering. *Requirements Engineering*, 13(2):87–115, 2008.

[79] Ivan J Jureta, John Mylopoulos, and Stephane Faulkner. Revisiting the core ontology and problem in requirements engineering. In *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 71–80. IEEE, 2008.

[80] Daniel Kahneman and Amos Tversky. Prospect theory: An analysis of decision under risk. *Econometrica: Journal of the Econometric Society*, pages 263–291, 1979.

[81] Nikos Karacapilidis and Dimitris Papadias. Computer supported argumentation and collaborative decision making: the hermes system. *Information systems*, 26(4):259–277, 2001.

[82] Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, 39(14):939–947, 1998.

[83] Paul Kay and Willett Kempton. What is the Sapir-Whorf hypothesis? *American Anthropologist*, 86(1):65–79, 1984.

[84] John Krogstie, Odd Ivar Lindland, and Guttorm Sindre. Towards a deeper understanding of quality in requirements engineering. In *Advanced Information Systems Engineering*, pages 82–95. Springer, 1995.

[85] Werner Kunz and Horst WJ Rittel. *Issues as elements of information systems*, volume 131. Institute of Urban and Regional Development, University of California Berkeley, California, 1970.

[86] Bryan Lawson. *How designers think: the design process demystified*. Routledge, 2006.

[87] Jintae Lee. Extending the potts and bruns model for recording design rationale. In *Software Engineering, 1991. Proceedings., 13th International Conference on*, pages 114–125. IEEE, 1991.

[88] Jintae Lee and Kum-Yew Lai. What's in design rationale? *Human–Computer Interaction*, 6(3-4):251–280, 1991.

[89] Julio Cesar Sampaio do Prado Leite and Peter A Freeman. Requirements validation through viewpoint resolution. *Software Engineering, IEEE Transactions on*, 17(12):1253–1269, 1991.

[90] Emmanuel Letier and Axel Van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 53–62. ACM, 2004.

[91] Sotirios Liaskos, Sheila A Mcllraith, Shirin Sohrabi, and John Mylopoulos. Integrating preferences into goal models for requirements engineering. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 135–144. IEEE, 2010.

[92] Sarah Lichtenstein and Paul Slovic. *The construction of preference*. Cambridge University Press, 2006.

[93] Panagiotis Louridas and Pericles Loucopoulos. A generic model for reflective design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(2):199–237, 2000.

[94] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[95] Matthew McGrath. Propositions. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[96] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In *Argumentation in artificial intelligence*, pages 105–129. Springer, 2009.

[97] Dennis C Mueller. *Public choice: an introduction*. Springer, 2004.

[98] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering, IEEE Transactions on*, 18(6):483–497, 1992.

[99] John Neter, William Wasserman, and Michael H Kutner. Applied linear regression models. 1989.

[100] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering, IEEE Transactions on*, 20(10):760–773, 1994.

[101] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[102] Eric Pederson, Eve Danziger, David Wilkins, Stephen Levinson, Sotaro Kita, and Gunter Senft. Semantic typology and spatial conceptualization. *Language*, pages 557–589, 1998.

[103] Maria Silvia Pini, Francesca Rossi, Kristen Brent Venable, and Toby Walsh. Aggregating partially ordered preferences. *Journal of Logic and Computation*, 19(3):475–502, 2009.

[104] John L Pollock. Defeasible reasoning. *Cognitive science*, 11(4):481–518, 1987.

[105] Henry Prakken and Gerard Vreeswijk. Logics for defeasible argumentation. In *Handbook of philosophical logic*, pages 219–318. Springer, 2002.

[106] Balasubramaniam Ramesh and Vasant Dhar. Supporting systems development by capturing deliberations during requirements engineering. *Software Engineering, IEEE Transactions on*, 18(6):498–510, 1992.

[107] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, 2001.

[108] Nicholas Rescher. The logic of preference. In *Topics in Philosophical Logic*, pages 287–320. Springer, 1968.

[109] Filippo Ricca, Giuseppe Scanniello, Marco Torchiano, Gianna Reggio, and Egidio Astesiano. On the effectiveness of screen mockups in requirements engineering: results from an internal replication. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 17. ACM, 2010.

[110] Horst WJ Rittel and Melvin M Webber. Dilemmas in a general theory of planning. *Policy sciences*, 4(2):155–169, 1973.

[111] William N Robinson, Suzanne D Pawlowski, and Vecheslav Volkov. Requirements interaction management. *ACM Computing Surveys (CSUR)*, 35(2):132–190, 2003.

[112] Stewart Shapiro. Classical logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2013 edition, 2013.

[113] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[114] S Buckingham Shum. Design argumentation as design rationale. *The encyclopedia of computer science and technology*, 35(20):95–128, 1996.

[115] Simon Buckingham Shum and Nick Hammond. Argumentation-based design rationale: what use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, 1994.

[116] Guillermo Simari and Iyad Rahwan. Argumentation in artificial intelligence. 2009.

[117] Guillermo R Simari and Ronald P Loui. A mathematical treatment of defeasible reasoning and its implementation. *Artificial intelligence*, 53(2):125–157, 1992.

[118] Guttorm Sindre and Andreas L Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[119] Barry Smith and Christopher Welty. Ontology: Towards a new synthesis. In *Formal Ontology in Information Systems*, pages 3–9. ACM Press, USA, pp. iii-x, 2001.

[120] Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer, 2010.

[121] Mark Staples. Critical rationalism and engineering: ontology. *Synthese*, pages 1–25, 2014.

[122] Chris Starmer. Developments in non-expected utility theory: The hunt for a descriptive theory of choice under risk. *Journal of economic literature*, pages 332–382, 2000.

[123] Masaki Suwa, John Gero, and Terry Purcell. Unexpected discoveries and s-invention of design requirements: important vehicles for a design process. *Design Studies*, 21(6):539–567, 2000.

[124] Barbara G Tabachnick, Linda S Fidell, et al. Using multivariate statistics. 2001.

[125] Alfred Tarski. The semantic conception of truth: and the foundations of semantics. *Philosophy and phenomenological research*, 4(3):341–376, 1944.

[126] Richard Thaler. Toward a positive theory of consumer choice. *Journal of Economic Behavior & Organization*, 1(1):39–60, 1980.

[127] Amos Tversky and Daniel Kahneman. Judgment under uncertainty: Heuristics and biases. *science*, 185(4157):1124–1131, 1974.

[128] Amos Tversky, Paul Slovic, and Daniel Kahneman. The causes of preference reversal. *The American Economic Review*, pages 204–217, 1990.

[129] JFAK van Benthem and Alice Ter Meulen. *Handbook of logic and language*. Elsevier, 1996.

[130] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[131] Axel Van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *Software Engineering, IEEE Transactions on*, 24(11):908–926, 1998.

[132] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, 2000.

[133] Achille Varzi. Mereology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[134] Douglas N Walton. *Informal logic: A handbook for critical argumentation*. Cambridge University Press, 1989.

[135] Philippe Weil. Nonexpected utility in macroeconomics. *The Quarterly Journal of Economics*, pages 29–42, 1990.

[136] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty HC Cheng, and J-M Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009.

[137] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[138] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[139] Eric SK Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

[140] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

[141] Milan Zeleny and James L Cochrane. *Multiple criteria decision making*, volume 25. McGraw-Hill New York, 1982.

[142] Jiajie Zhang. The nature of external representations in problem solving. *Cognitive science*, 21(2):179–217, 1997.