Requirements Modelling Language Design

—

How to Make Formalisms for Problem Solving in Requirements Engineering

—

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

February 23, 2015

Outline

You have a Requirements Engineering Problem to solve, if (i) you have unclear, abstract, incomplete, potentially conflicting information about expectations of various stakeholders, and about the environment in which these expectations should be met, (ii) you know that there is presently no solution which meets these expectations, and (iii) you need to define and document a set of clear, concrete, sufficiently complete, coherent statements, often called requirements, so that a system which is made and operated to satisfy these statements will in fact meet stakeholders' expectations.

Requirements Engineering Problems are ill-structured problems. Solving them is hard. You typically have to do many complex and interdependent tasks, such as elicitation, categorisation, evaluation, prioritisation, negotiation, prediction, and so on.

Solving a Problem mobilises intuition and creativity, but also experiential knowledge that comes from your own and others' problem solving experience. This book shows how to make Artificial Intelligence (AI) from experiential problem solving knowledge. You can see such AI as an assistant, to which you can delegate some problem solving tasks, while you focus on others.

AI are combinations of formal languages, used to represent (make models of) problem solving information, and algorithms used to do transformations of, and computations on those representations. Languages and algorithms preserve and automate the application of the experiential problem solving knowledge which you chose to build into them.

ACM Classification keywords: Requirements analysis, Formal language definitions, Context specific languages, Formalisms.

About the Author

Ivan Jureta is Chercheur qualifié (tenured researcher) with Fonds de la Recherche Scientifique – FNRS in Brussels, and Associate Professor with the Department of Business Administration, University of Namur.

Ivan's research focuses on how to identify and elicit knowledge which human experts apply to solve ill-structured problems, how to engineer ontologies and processes which capture and preserve that knowledge, and how to engineer artificial intelligence which automatically applies that knowledge at a larger scale. This interest falls within the various research fields concerned with the transfer, preservation and automation of knowledge.

Ivan is the author of "Analysis and Design of Advice" (Springer, 2011), and has published over 60 research papers on these topics, many of which focus on knowledge applied when engineering new software. He has published in the fields of requirements engineering, business analysis, and conceptual modelling of information systems, and serves on scientific committees of the International Conference on Advanced Information Systems Engineering (CAiSE), the International Conference on Conceptual Modelling (ER), and the International Requirements Engineering Conference (RE). He is the recipient of the 2008 IEEE RE conference best paper award, and the 2014 CAiSE conference distinguished paper award. He has organised and chaired the series of International Workshops on Modelling and Reasoning for Business Intelligence (MORE-BI), held in Brussels in 2011 and Florence in 2012.

In parallel, Ivan is involved with several startups at CxO level and have held lead roles in product design for web and digital services that today serve more than 500.000 people every day.

Acknowledgements

This book was influenced by research which I did with John Mylopoulos and Alexander Borgida. I co-authored papers on Requirements Modelling Languages with them, as well as with Neil Ernst, Stéphane Faulkner, Anna Perini, Pierre-Yves Schobbens, Alberto Siena, Angelo Susi, and many others. They have all shaped the content of this book in various ways. This does not mean that we agree on the ideas which I present here.

Copyright

Contents

List of Figures

Chapter 1

Requirements Problem Solving

This Chapter clarifies what Requirements Problem Solving is. Section 1.1 gives reasons why it is interesting to study Requirements Problem Solving and learn how to make Artificial Intelligence which automates tasks of Requirements Problem Solving. Sections 1.2 to 1.4 give the characteristics of Requirements Problem Solving, by describing the Problem Situations which initiate Requirements Problem Solving, tasks done in Requirements Problem Solving, and the Solution Situations sought by doing these tasks. Section 1.5 gives a synthetic definition of Requirements Problem Solving.

1.1 Motivation

This Section argues why Requirements Problem Solving is interesting to study, and why it is relevant to make Artificial Intelligence which automates tasks in Requirements Problem Solving.

This book is about how to make Artificial Intelligence that helps you solve Problems. If your expertise is in, or is related to these topics, then this probably brings to mind many ideas on notions such as ontologies, formal languages, algorithms, complexity, and so on. They will turn up later.

In non-technical terms, this book is about how to define specific kinds of instructions which can be given to computers. This is done so that you can delegate to them some of the work done when solving a particular kind of common, but hard problems.

Consider the following situations. To successfully get out of each, you have to solve a problem. All these problems share some common characteristics, and because of this belong to the class of so-called Problems.

- A law firm has a new owner. She saw that the employees are delivering the same services in different ways, and believes these inconsistencies will inevitably lead to lower quality in the future. She asks you to suggest how to avoid this in the long term.

- A company makes software which sports coaches use to give training instructions to athletes. The owner wants that every future software improvement helps reduce the time that coaches spend on repetitive tasks. You are asked how to ensure this.

- A company makes software products for telecommunications service providers. The company CEO wants that all its current and future software products have clearly defined rules and processes for customisation, installation, maintenance. He asks you to propose how to ensure this in the long-term.

- A small firm designs its products and outsources manufacturing and distribution to other companies. The managing director is interested in investing in software, which would help keep track of the status of new product development tasks. You are asked to suggest how to proceed.

- An electronics manufacturing company sells its products through distributors. A regional marketing director wants to improve sales decisions by collecting more merchandising data, that is, data about product placement in distributors' points of sale. You are asked for advice on how to do this.

These situations are different. They are about *different* industries, companies, products. They involve different people, and consequently not the same expertise, experiences, or expectations. In reality, they also happened to me at different times and places, over the course of a few years.

The obvious question is:

What would you do to successfully resolve each of these situations?

The motivation for this book *does originate* in the need to answer this question. However, the book's focus is *not* to answer it. As I will mention later, there are several fields of research and practice which already provide ample material about what to do in these situations.

Instead, the book aims to answer the following question:

If you do know how to successfully resolve situations such as the above, and are often in such situations, then how can you delegate some of your problem solving effort to computers?

There are several simple reasons to want to answer this question. If computers do part of the work, you have more time to apply intuition and creativity to the rest of problem solving, or for solving other problems. Others can use these same computers, or more precisely software, when in similar situations on their own. Finally, the instructions you define, and which the computers apply, constitute a record of parts of your problem solving knowledge, which may be a relevant source of learning for others.

1.2 Problem Situations

This Section lists the characteristics of Problem Situations which trigger Requirements Problem Solving.

I briefly described five situations in the preceding section. Each gave cues to there being problems, but it gave no clear, complete, precise, and coherent description of each problem. This absence of a

given problem is one of several characteristics of situations, which I focus on in this book. There are Problems to solve in these situations.

The interesting situations, called Problem Situations hereafter, have the following characteristics.

1. *The problem needs to be defined. It is not a given.* Instead, there is information about what someone wants and believes about the situation, the reasons and causes behind it. There is nothing that guarantees that realising what they say they want, will indeed alleviate the troubles observed in that situation.

   In absence of the exact and precise problem to solve, you have to find it out, describe it clearly, communicate it to those who expect the solution, and have them agree that solving that problem, rather than another, will make them happy.

   For example, the owner of the law firm may know what she dislikes about the current situation. But that tells you nothing about the events which led to that situation, and the actions which led to these events. The problem may be, for instance, that current instructions and incentives allowed and motivated these actions.

2. *Defining the problem requires collaboration.* A given situation may involve one or many different problems, which may be independent or somehow related. There is rarely one clear problem that you can observe and describe in isolation from everything else. I emphasise that the problem is defined, and the parties involved need to agree that this is indeed the problem to solve.

   The information you may need to define the problem is not necessarily held by one individual. Problems may arise in situations, or cause situations, which involve people with different backgrounds, expertise, and expectations, and it may be that many of them have information which you need to define the problem.

   The regional marketing director of the electronics manufacturer may want to collect some merchandising data. It may be that only the distributor's lawyer knows if giving that data clashes with the distributor's contracts with other manufacturers.

To find the with repetitive tasks that sports coaches do, and which the software should help them with, makes it necessary to collaborate with these coaches, on identifying and understanding such tasks, with engineers to understand the feasibility of automating these tasks, with the software product designer to prioritise which tasks the software should automate, in which release, and so on.

3. *Different people see different problems.* Different people involved in a given situation perceive the problem or problems in it differently, depending on their expertise, experience, motivations, and so on. In the same situation, a management professional may focus more on coordination between people, a lawyer on the relationships between actions and applicable legal constraints, a finance professional will look for incentive problems, and so on. Not all perspectives always matter, but it may also be that defining the problem by focusing on one of them only will lead to irrelevant solutions.

   You might think the law firm owner's problem as one of leadership. An employee of the same firm may see it as an incentive problem. A information technology specialist could conclude that the situation is caused by the absence of relevant software.

1.3 Problem Solving Tasks

This Section lists the characteristics of tasks done in Requirements Problem Solving.

What to do in a Problem Situation? A number of different tasks can be done. The overall aim is to move from the situation in which you recognised that there is a problem, to a situation in which the relevant parties, **stakeholders** hereafter, recognised that the problem no longer exists. All that happens between these two points is called problem solving.

There are different kinds of problem solving tasks. They require different skills and inputs, give different outputs, and can be done at different times during problem solving. The following are typical kinds of problem solving tasks done for Problems. I cite some of the relevant research on each.

- *Elicitation* are all tasks done in order to obtain information from people, or any other source. Collection and analysis of documentation, interviews, workshops, observation of stakeholders or others, are some of possible elicitation tasks [56, 71, 38].

- *Representation, or modelling* are tasks that aim to divide available information, about the problem or its solutions, into pieces that share similar characteristics, and, or which somehow make a coherent whole, and to represent these pieces in models, which in turn are used to draw conclusions useful for deciding what to do next in problem solving [36, 155, 88].

- *Clarification* involves identifying, for example, ambiguous, vague, or otherwise unclear information, and finding out more in order to repair these deficiencies, if they are deficiencies in the first place [108, 99, 85].

- *Prioritisation* consists of deciding what the relative importance of various sub-problems is, so as to better use the available resources to work on dedicating more resources to work on the most pressing ones. It also means deciding which solution parts to implement before others [91, 11, 70].

- *Negotiation* aims to reconcile disagreements about the problem and its solutions among the stakeholders [98, 13, 82].

There are other problem solving tasks, and they include responsibility allocation [36, 23, 51], cost estimation [14, 17, 131], conflicts and inconsistency [110, 69, 145], comparison [108, 99, 100], satisfaction evaluation [16, 108, 93], operationalization [54, 51, 47], traceability [57, 118, 31], and change [26, 149, 20].

Almost all kinds of problem solving tasks mentioned above are *not* specific to solving Problems. Medical diagnostics involves, for example, elicitation from the patient, via interview and medical devices, clarification to the patient, prioritisation of symptoms and potential conditions, negotiation in case the patient disagrees with the treatment, responsibility allocation to the patient to take medications or medical personnel to administer treatment, comparison of possible treatments for the same condition, change of treatment in case it proves inadequate.

The following characteristics of problem solving for Problems sets them to some extent apart from other classes of complicated problems.

1. *Problem definition and solution definition are intertwined.* In simple terms, since you are defining the problem to solve, you are also defining what its solutions can be. Pushing this idea to its extreme, perhaps you can convince the stakeholders that there is no problem, which is a solution. The other extreme is that you are defining an unsolvable problem, within any, or the given resources. Most cases are between the two, where the problem you define is influenced and influences the expectations of the stakeholders, and restricts the range of possible solutions that can be considered, and the extent to which their expectations can be met.

   If, for instance, you convince the law firm owner that the problem is something intangible and abstract as employees react by being destructive to her excellent leadership style, then potential solutions narrow down to, for example, firing everyone, which would likely close the law firm, given that it will become impossible to serve the same clients with untrained new employees.

2. *Problem definition identifies new (sub-)problems.* There is no guarantee that defining one problem will not lead you to find others. Defining the problem in one Problem Situation can lead you to another.

   In the company which makes software products for telecommunication service providers, elicitation from an employee experienced in product customisation may tell you that customisations take time to define clearly with clients, and that shorter times are not feasible. Elicitation with the CEO may suggest that this should take considerably less time. The clash between the two pieces of information may be a symptom of a problem of there being few people, with little time to actually talk to clients to define customisations. It may be that clients take time to decide. Both are new problems, which may need to be solved, in order to proceed to the solution of the initial problem.

3. *Problem solving rarely stops because the best solution is defined.*

A solution changes the initial Problem Situation. Even elicitation changes it, since your interaction with stakeholders can change what they understand the problem to be. Events occur, which may or may not be under your or stakeholders' influence. They can change stakeholders' understanding of the Problem Situation, of what worked or did not, and why you got involved in the first place.

Change means that the best, optimal solution is a moving target. Problem solving thus stops more due to resource constraints, than to the willingness to seek the best solution, or more generally, to move from the Problem Situation, to one which is more desirable.

In the law firm, elicitation involved questions about when and what caused errors in the work with clients. These questions alone create expectations with the employees, who expected some inevitable change in the organisation, be it training, changes of work processes and rules, firing, and so on. This in turn created anxiety, which contributed to errors observed in work with clients, which were observed during problem solving. Changed work practices and rules in turn generated new anticipated problems, creating new Problem Situations, and requiring problem solving to continue.

1.4 Solution Situations

This Section lists the characteristics of Solution Situations when Requirements Problem Solving stops.

A Solution Situation is when problem solving can stop. This can be a hard target to aim for, given that solutions to Problems tend to share the following characteristics.

1. *Solutions are temporary.* Change in the environment and stakeholders' expectations will make you move from one Problem Situation to another, requiring moves from one Solution Situation to another.

   Sports coaching is not a well-defined, and unchanging activity. It changes as coaches learn new knowledge about the mechanics of the body, about the psychology of the athletes, about the relationship of these to environment conditions. Automating

some of the tasks in sports coaching is consequently a task which can go on indefinitely, or at least for a very long time, as a perfect universal sports coaching method is a long way off, or simply does not exist.

2. *Best Solutions are elusive.* Given that different people can see different problems in the same Problem Situation, that problems cannot be treated in isolation, and that conditions change, optimal solutions are unlikely to be easy to find, or may not exist at all. Good enough solutions have to do instead. Good enough in practice means that at least some of the most important expectations of the stakeholders are satisfied.

3. *Defining, making, and running a solution requires collaboration.* Agreeing that the solution should, for example, involve the introduction of new work rules and processes, requires designing these, training people to use them, perhaps making software that automates some, and responding to issues that arise, once the processes and rules are used. This can require expertise from management professionals, organisational psychologists, human resources specialists, business analysts, software and hardware engineers, technical support specialists.

1.5 Requirements Problem Solving

This Section gives the definition of Requirements Problem Solving used throughout this book. The definition does not list activities done when doing Requirements Problem Solving. Instead, it says that Requirements Problem Solving includes all activities triggered in response to situations which satisfy specific properties. The definition lists these properties.

The characteristics of Problem Situations and Solution Situations presented in this Chapter suggest that there is no known best way to do Requirements Problem Solving. It is not known exactly which problem solving activities need to be done, when one starts and the other stops, the exact properties their inputs and outputs should have, and the steps to take to collect the required inputs, and to produce the desired outputs.

As I will argue in Chapter 2, Requirements Problem Solving is a kind of ill-structured problem solving, an argument which will further support observations in the paragraph above.

In conclusion, an exact and relevant list of activities that make up Requirements Problem Solving is elusive. A definition of Requirements Problem Solving via such a list is consequently also elusive.

My suggestion is to define Requirements Problem Solving as all activities that people do, when confronted to Problem Situations that satisfy specific properties or rules. These rules are not as elusive.

Requirements Problem Solving

Definition 1.5.1. Requirements Problem Solving are all activities done in response to situations, called Problem Situations, which satisfy the following conditions

1. there are unclear, abstract, incomplete, potentially conflicting information about expectations of various stakeholders, and about the environment in which these expectations should be met,

2. there is presently no clear definition of the problem to solve, in order to satisfy these expectations, and no solution which indeed meets them, and

3. it is necessary to define and document a set of clear, concrete, sufficiently complete, coherent statements which

   (a) define the problem that stakeholders agree on, and

   (b) define the solution that the stakeholders agree on, such that if the solution is made and used, then it should in fact meet stakeholders' expectations, and thereby solve the problem.

The resulting definition of Requirements Problem Solving places emphasis on conditions which trigger it. It suggests that Requirements Problem Solving includes all activities done in response to these triggers. The definition is therefore not prescriptive, suggesting what Requirements Problem Solving should be, or when it is good or bad. Requirements Engineering is an engineering discipline and field of research which prescribes which activities to do, a topic I develop in Chapter 3.

Chapter 2

Problem Solving Automation with Artificial Intelligence

This Chapter connects Requirements Problem Solving with central ideas and terminology of problem solving and Artificial Intelligence. Section 2.1 clarifies what automation means in this book, and why it is interesting to automate tasks in Requirements Problem Solving. Section 2.2 argues that Requirements Problem Solving is one type of ill-structured problem solving, which makes it impossible to automate fully, and recalls the characteristics of ill-structured problems. Section 2.3 recalls the characteristics of well-structured problems, whose resolution can be automated, and Section 2.4 argues that there are well-structured sub-problems that can be identified in Requirements Problem Solving. Section 2.5 argues that there are tasks in Requirements Problem Solving, which reappear across cases, and automating them is particularly interesting. Section 2.6 connects automation of recurrent Requirements Problem Solving tasks to the concepts of formal language and algorithm. Section 2.7 connects these ideas to the basic terminology of Artificial Intelligence.

2.1 Automation

This Section clarifies what automation means in this book, what its purpose is in Requirements Problem Solving, and gives an outline of ideas developed in the rest of the Chapter.

AI's role in this book is practical. It is used to automate problem solving tasks. I will say that a task is automated if it can be done by a computer. The specifics of the computer, its speed, memory, or other characteristics, are not relevant for now.

There are machines which automate tasks without being computers. For example, a mechanical arm which closes a door. I am interested specifically in computers and not such machines, because Requirements Problem Solving tasks involve taking and making changes to data. There is no need for the mechanical arm to have a representation of the door being open or closed, or of the door itself, and do computations on those representations, in order to accomplish closing the door. In Requirements Problem Solving, there is a need to have data about phenomena in Problem Situations and Solution Situations, and to do computations on it. The AI for Requirements Problem Solving may recommend through that data what to do next, but I will assume that people who read this data remain autonomous to choose themselves the course of action.

Benefits of automation should be obvious, and I already mentioned them earlier. If a machine can do part of the problem solving work, you can invest it in improving the quality of the solution to the current problem, or on improving your overall efficiency in problem solving (by solving more problems in the same amount of time).

Artificial Intelligence is the means to automate *some* Requirements Problem Solving tasks. It cannot automate all of them, because it is still not clear which they are and how exactly to do them. Many of them rely on intuition and creativity. For example, it is not clear today how to automate elicitation via interviews, negotiation between stakeholders, or the creation of relevant representations of what stakeholders say, to name some. All this might be amenable to automation using some advanced, and currently unknown forms of AI, a topic which is speculative and remains outside the scope of this book.

To explain the practical role AI has in Requirements Problem Solving, I will present and argue for the following ideas and in the given order. Each has its own section in the rest of this chapter.

1. *Requirements Problem Solving is one of many kinds of ill-structured problem solving.* This is important, because if Requirements Problem Solving is a kind of ill-structured problem solving, then I am right to argue that Requirements Problem Solving cannot be fully automated.

2. *It is possible to identify tasks in Requirements Problem Solving, in which the sub-problems to solve are not ill-structured, but well-structured, in that the tasks for solving them can be automated.* In other words, although Problems are ill-structured as a whole, there is some structure to some of their parts, and such parts can be treated as well-structured problems.

3. *To automate the solving of a well-structured sub-problem, it is necessary to have algorithms which solve it, and in turn, to have rules for how to communicate with these algorithms.* You need to communicate the data about Problem Situations and Solution Situations, and the algorithms should be able to communicate back data in a way which is understandable to human problem solvers. If this is the case, then it is relevant to have specialised languages for communicating with the algorithms.

4. *AI for Requirements Problem Solving amounts to combinations of algorithms that automate problem solving tasks, and languages for communicating with these algorithms.*

2.2 Ill-Structured Problems

This Section argues that Requirements Problem Solving as a class, or type of activities falls within a broader class of activities called ill-structured problem solving. Characteristics of ill-structured problems are given, so that you can compare them to those of Requirements Problem Solving.

You can see this by comparing the characteristics I gave for Requirements Problem Solving, and the characteristics of ill-structured problem solving, which I give in this section. By Requirements Problem Solving characteristics, I mean those of Problem Situations, problem solving tasks, and Solution Situations in Chapter 1.

There are two main reasons why it is important to be concerned with whether Requirements Problem Solving is a kind of ill-structured

problem solving. Firstly, it influences the perception of how much of Requirements Problem Solving can be automated. Secondly, it is relevant to know if Requirements Problem Solving is something entirely new, unrelated to existing research and practice, or an old discussion topic.

Given the similarities between the characteristics of Requirements Problem Solving and ill-structured problem solving, my first conclusion is that Requirements Problem Solving cannot fully be automated. Only some parts of it can, as I will argue in Section 2.3. My second conclusion is that Requirements Problem Solving is neither a fundamentally new kind of problem solving, nor targets a new class of problems. However, Requirements Problem Solving is a hard activity to do well and it is very relevant to research how to better do and automate Requirements Problem Solving. At least three decades of existing research agree on this, as I will mention in Chapter 3.

The rest of this section recalls the well-known characteristics of ill-structured problem solving, specifically in engineering domains. I give these characteristics so that you can decide if you agree with my conclusions above.

To illustrate the characteristics, I borrow quotes from interviews that Jonassen, Strobel, and Lee [81] did with engineers experienced in solving ill-structured problems.

Workplace problems are often ill-structured, including in engineering domains, in that information is missing, there is a need to work and coordinate with other people to get and clean up the information, there is a need to negotiate, to reach agreement, and so on.

> *"Probably the biggest challenge that we see in some of these projects is dealing with incomplete information. Invariably people won't know what the output is going to be for the product. So you don't know what kind of cooling load or heating load [is] to be expected. You don't know the specific heat is because its not listed. Or any number of design parameters that are not defined. In some cases you are making assumptions in design and you're making critical assumptions that you can do what you're wanting to do based on some piece of information or lack of information."*

Ill-structured problems include well-structured sub-problems, as

there may be sub-problems which, if properly formulated, can be solved with existing tools and algorithms.

> *"We had to decide how big [to build] a lagoon to hold the dirt and the possible rain for the possible amount of time we were treating this soil. And we had to decide how best to treat the soil. We had to make calculations how long [it] would take to put back into the ground. As we were doing all this [we had to] decide where we would sample the soil and separate [it]. We had to figure out what we were going to find in the hole, how we would treat it so what we would know about the size, and what to put in the water treatment system, and how we were going to power it. So those were a few of the decisions we needed to make.*
>
> *With all the data that we collected out in the field on the performance of them in the past years, we looked at which had the fewest cracks, which had come loose the most, which had the fewest repairs, and which were more impermeable to chlorite, to salt that gets down in them and corrodes the rebar. It was analysing the data and then also trying to confirm that with me other state DOTs that had used the same thing."*

Ill-structured problems often involve many potentially conflicting goals, meaning that even when there is one seemingly clear main goal, such as "build new terminal to increase the capacity of the airport", there are sub-goals, or equally important goals which can come from other stakeholders, such as "keep the number of flights constant or lower, in order to maintain or reduce noise pollution". Here is an example from a construction project, still from Jonassen, Strobel, and Lee [81]:

> *"We'll measure it with a variety of things. Number 1, did we meet the anticipated goals for hiring of a diverse work group. That is part of the contractual requirements as well as the participation of a variety of different kinds of enterprises. Our safety record, and of course, did we make any money on the project? [...] Our goal is always to work safely and make money. Safely and on time and make our clients happy and to get additional work from our client."*

There can be many different, and initially unknown ways to solve an ill-structured problem, so that the best approach is normally not known, and has to be designed from scratch, or found among known alternative approaches.

The success of a solution is rarely measured by engineering standards, as it involves criteria such as satisfying the client, on-time completion, staying within the budget, as well as various legal, regulatory, and other.

> *"So we are pretty savvy as to understanding what the code is trying to say. You have some people in these code making bodies that you can consult to make formal interpretations and writen interpretations. So we had to make sure the bank would give us a line of credit and we had to talk to our client to see if they would pay us some up front money to start building these systems. Funding was a big concern, and we have to make sure we have legal constraints as long as we are complying with the law, and we want to make sure the client is not asking us to do something illegal. We also want to make sure that we have a contract where every party is happy, if that is possible. Make sure we get paid, and they understand what we are going to do and we understand what their expectations are."*

Many constraints in problem solving are unrelated to engineering, such as cultural, political or environmental ones.

> *"We are solving a whole series of things in the fact that an architect or owner wants to build a building a certain way, and he has certain needs and desires, but yet we have all these safety codes that need to be met...the state...building and Fire Department that had their whole set of requirements. And we had to make sure that those requirements also didn't pose problems. So we had a whole series of requirements one playing off against the other that we had to balance out. And there were several environmental issues up there, concerns from the U.S. Fish and Wildlife that we were going to hurt the fish."*

Knowledge required to solve the problem is distributed across different people, in same or different, related or unrelated organisations, and they all have to contributed during problem solving.

> *"There's certainly the property owner, there's the telephone company regional manager, there's the utility company chief engineer, there's the utility company distribution engineer, there's there utility company attorney. There's the utility company's insurance company attorney. There's the homeowners, the homeowners' insurance company attorney. There's his insurance adjuster. There's a fire investigator, two fire investigators, that's about all, there's my electrical testing company technicians.*
>
> *Inside our organization engineers, partners, cad drafters, graphical, computer, secretarial help. Outside our form we interfaced with the architect, the owner, the structural engineer, mechanical engineer, electrical engineer. We interfaced with all those disciplines because it is essential to have all those things working together as a package."*

Problem solving requires extensive collaboration, which is not unusual, given that the problem solving knowledge is distributed.

> *"We all pretty much know our roles but know that in our specialisation those people touch on certain things affect fire protection engineering and life safety.*
>
> *We are all working together for a common goal, which is to make sure that we have an economically viable building and a safe building – on that is going to function the proper way. We all sit down at the conference table together and we come up with a plan and then we work very closely with the engineering disciplines so we have all the details ironed out."*

Experience guides problem solving, in that they will rely on information that they can recall from the past, when they faced and did, or failed to solve similar problems, and do so more than rely on their understand of the abstractions which the concrete problem instantiates.

> *"Experience some problems like [those] that have occurred in the past. Experience on those things is probably the biggest way we get them solved quickly anyway."*

Unanticipated problems often arise in problem solving, as the environment and expectations may change, and more specifically budgets, regulations, client's goals, and so on.

"Also at different times we don't live in a perfect world and when buildings get put together at times people can make mistakes. Sometimes they can't be rectified and need to be ripped out and other times where it would be disastrous to do that so we develop equivalency concepts for that. The other unanticipated thing is you can get in a project and the owner can change his mind and all of a sudden the whole dynamics of the project changes."

2.3 Well-Structured Problems

This Section recalls the characteristics of well-structured problems. This is relevant, because the solving of well-structured problems can be automated.

A problem is well-structured if there is sufficient knowledge about how exactly to describe it, what its solution is, and what exactly to do, in order to reach the solution. This is reflected in the characteristics of well-structured problems, which I recall below. They are essentially the same as those that Herbert Simon argued for, and which seem to be widely accepted in artificial intelligence research [130].

(1) Well-defined Solution Situation: There is a well-defined situation, called Solution Situation, which if observed, means that the problem is solved.

A well-defined situation means thaty there is a set of variables, and each obtains a value. This is also called a solution state. I use the terms "situation" and "state" interchangeably. Variables can be "product profitability", "employee satisfaction", or something that has a widely accepted definition, such as "outside temperature", or "elevation".

Clearly defined also means that there is no room for interpretation of what variables and their values stand for. For example, $x = 5$ is clearly defined only if it is known how to measure, observe, compute, or otherwise obtain the value of x, in a way stakeholders agreed on.

A condition such as "product is profitable" is not clearly defined if stakeholders disagree about the exact way to measure product profitability. If, however, all agree about what exactly profitable means for that specific product (say, that if the accounting profit generated through sales of a product, is above a given threshold amount of money), then it is clearly defined.

(2) Solution test: There is a test, called solution test, which can be done to check if the Solution Situation is reached. If the Solution Situation is, for example, that "employees are satisfied with new work processes", there has to be a procedure to apply, in order to check if this is the case. To have the procedure, you have to know how to measure "employee satisfaction", and in absence of a universal definition (which is present if you are measuring, say, distance on land), you need an instrument for measurement, and the stakeholders have to agree on it. The instrument could be a survey questionnaire. The test, in turn, could be that you distribute the questionnaire to a sample of employees, or all of them, and observe a specific pattern of responses.

(3) Language: There is a set of agreed upon terms, and of rules for using these terms, for describing the Problem Situation, Solution Situation, and intermediate situations. Such languages will be called Requirements Modelling Languages in this book, as they are used to represent, or make models of information used during Requirements Problem Solving.

Take a simple example, such as a heating a room when you are cold. The thermometer reads 20 centigrade, the heater is on, and at heating intensity 2 (out of five intensity values). What I just wrote are the terms which define three variables, for temperature, heater being on, and heating intensity. Each is clearly defined, as you know the possible values, you know which of them you can directly change values of (heater being on, and heating intensity), and how to measure the one you cannot (temperature). The initial state of the problem is defined as that assignment of values to the variable, that is, temperature is equal to 20 centigrade, heater status is on, and heating intensity is 2. You decide that the Solution Situation is temperature being 25 centigrade. In an intermediate situation, you may measure that temperature is 22 centigrade.

In other words, there is an agreed on language for describing the initial conditions, in which there is the problem, the intermediary conditions that you get into by doing problem solving, and the solution that you want to reach.

(4) Operator: There is a set of operators used to change from one situation to another. For each operator, the conditions when it applies are known, that is, it is known which values variables have to have, in order to be able to apply the operator.

You can think of an operator as an action that can be taken when

specific conditions are satisfied (that is, some variables take some values), and which changes the values of some other variables.

Continuing the heating example, an operator is you changing the heating intensity level, by turning a knob on the heater. The operator applies when heater status value is on, and its effect can be that you reduce or increase heating intensity from the current, to any one of the five intensity values. Another operator may be that you can turn the heater on or off, and thus change the value of heater status variable to either of its two values.

(5) Difference tests: There are tests that can detect differences between values of variables in two situations. They are used to detect how applying operators changed the values of variables.

A difference test may consist of you measuring room temperature, that is, checking the temperature, to conclude that changing heating intensity from value 2 to value 4 increases the room temperature by 6 degrees centigrade.

(6) Difference rules: There are rules defining which operators to use, to reduce differences observed with difference tests.

A difference rule may say that if the difference between the temperature in Solution Situation and the temperature in the current state is between 2 and 4 degrees centigrade, then apply the operator to increase heating intensity level, and increase it to level 3.

2.4 Well-Structured Sub-Problems

This Section argues that although Requirements Problem Solving is ill-structured problem solving, it is possible to identify tasks in it, which target sub-problems having all the characteristics of well-structured problems. Such tasks in Requirements Problem Solving can be automated.

If you wanted to approach the law firm owner's case as if it was a well-structured problem as a whole, then can you describe Solution Situation? What is the language to use to describe the Problem Situation, intermediary situations, and the Solution Situation? What are the variables? What are the possible values of these variables? What operators are there to act in those situations? How would you test is you reached Solution Situation? Same questions arise, if you wanted to apply well-structured problem solving to all cases in Chapter 1.

The trouble with these questions, and the main reason why those

cases involve ill-structured problem solving is that there is no existing knowledge, in research or practice, which gives agreed on answers. There is no standard which prescribes what to do in those cases. There is no book which guides you step-by-step from the Problem Situation to a satisfactory Solution Situation.

It is possible, however, to identify parts of the ill-structured problem, for which you can answer the questions above in some satisfactory manner.

In the law firm owner's case, issues seems related to how employees deliver services. It is likely that solving them requires the help of employees. It thus seems useful to evaluate employee satisfaction at present and in the future. Their satisfaction matters, and it would be relevant not to judge it informally, but to have an evaluation procedure, which they and the owner agree on.

Is making and doing the survey to evaluate employee satisfaction a well-structured sub-problem in the law firm case?

There is no standard or scientific rule for this. The quality of a solution is ultimately evaluated by stakeholders. The solution itself is influenced by the problem that you defined during Requirements Problem Solving.

It follows that you have three options. In one, the sub-problem is well-structured, because it you have the knowledge required to address it as a well-structured problem. This means that you know a language, relevant operators, and so on, *and stakeholders agree that your approach is appropriate*. You would define and do the surveys, which would require that you define the variables whose values are measured via data collected in the survey, the survey questions, and so on.

A second option is that you do not know how to solve it as a well-structured problem, and so for you it is an ill-structured problem. But, you may know someone whom you can delegate it to, and for whom it is a well-structured problem.

It is important to keep in mind that the resulting well-structured problem is one of measuring employee satisfaction, not changing it. The change is not part of this sub-problem.

Take another example. One of the tasks many of the sports coaches from the earlier case do, is invoicing, and some outsource it to their accountants. Invoicing includes producing invoices at regular intervals. There accountants capable of doing this, and there is also software capable of doing this. It may thus be that producing

invoices is a well-structured sub-problem. And notice that while to produce an invoice is an ill-structured problem for a coach, it is not for an accountant, or for software designed to do it.

The third option is that the sub-problem is ill-structured, when neither of the two options above are feasible.

The first two options are interesting, because they mean that

> *you managed to divide the ill-structured problem into parts, you can call them sub-problems, or at the very least, you identified one sub-problem which you, or someone you choose, is capable of solving as a well-structured problem.*

Problem structuring is the tasks which aims to identify sub-problems, and understand the relationships between them, as well as their solutions. This task itself need not be a well-defined sub-problem. You may never manage to have an accurate map, a description of all the sub-problems, and of all their relevant relationships.

But while you recognise one or some well-defined sub-problems, the rest of the initial and overall ill-structured problem need not be as well-behaved. Simon described the challenge as follows [130]:

> *"Interrelations among the various well structured subproblems are likely to be neglected or underemphasized. Solutions to a particular subproblem are apt to be disturbed or undone at a later stage when new aspects are attended to, and the considerations leading to the original solutions forgotten or not noticed."*

One of the recurrent topics in Requirements Problem Solving research is how to document the decomposition of the ill-structured problem onto sub-problems, to record their relationships, and to evaluate how solving some sub-problem in one way influences the resolution of another sub-problem.

2.5 Case-Specific and Recurrent Tasks

This Section argues that Requirements Problem Solving involves case-specific and recurrent tasks. It is interesting to automate recurrent tasks which target well-defined sub-problems.

In the preceding Section, I used as examples two tasks that can be relevant to do, in two different cases. One is surveying the employees of the law firm, the other is invoicing in the sports coach software case. An important idea to keep in mind, is that producing invoices and doing surveys are tasks specific to the Problem Situation. They are *case-specific* tasks.

It is interesting to develop knowledge of how to solve these sub-problems as well-structured problems. If you manage to do so, perhaps you can go further and automate the solutions. The result might be relevant to companies and, or individuals who happen to encounter similar Problem Situations.

This book focuses instead on the automation of *recurrent tasks* in Requirements Problem Solving, of tasks which reappear in different cases.

Problem structuring is one example of such a task. It is hard to avoid it when solving Requirements Problem Solving, regardless of case specifics. Keeping track of sub-problems and their relationships is not specific to a given Problem Situation, but seems necessary for any ill-defined problem, that is, whenever you do Requirements Problem Solving. It is therefore particularly interesting to try to make AI which helps with these recurrent Requirements Problem Solving tasks.

2.6 Languages and Algorithms

This Section argues that automating a task in Requirements Problem Solving requires defining formal languages, used to represent information about Problem Situations, Solution Situations, and intermediary situations, and algorithms which perform computations over these representations.

To have a computer automate a recurrent Requirements Problem Solving task, such that the task solves a well-defined sub-problem, you need to identify the sub-problem, define all the components required for it to be a well-structured sub-problem, and have them in a format which the computer can store, read, and do computations on.

This book is about how to do the above. Using simpler, but still technical terms, the challenge is to define languages and algorithms,

which together can be used to make software to which you can delegate a recurrent task in Requirements Problem Solving.

I will not go so far as to show how to actually make that software. Instead, I will stop at the point where it is clear what that software would do, and how.

Languages need to be formal languages, that is, sucht that terms and rules for using terms of the language are precisely defined. Such languages are used to describe situations, be they Problem Situations, Solution Situations, or intermediary ones.

Algorithms describe procedures applied to descriptions of situations. Algorithms will be doing calculations over variables that describe situations. They will takes some input written in the language, do computations (value calculations, changes, additions, deletions, etc.) to these, and produce results which are, again, written with the language, and describe situations.

You can see the algorithms as combinations of operators, solution tests, difference tests, and difference rules, all used when solving a well-structured sub-problem.

2.7 Artificial Intelligence

This Section argues that AI for Requirements Problem Solving amounts to combinations of a formal language and algorithms.

AI is a science on its own, and this book makes no contributions to it. Instead, I will use existing results in AI when they are useful to automate Requirements Problem Solving tasks.

Making AI for Requirements Problem Solving in this book equates to the problem of how to make languages and algorithms which can automate specific Requirements Problem Solving tasks. Both the languages and algorithms are not somehow made by, or discovered by the AI, but at best imperfectly reflect the knowledge of human problem solvers. The resulting AI is thus not smart in any meaningful way, but is only able to apply quickly many rules which are inspired by what an expert problem solver would otherwise do.

In the rough classification of major lines of research in AI, which Stuart Russell and Peter Norvig suggested in a classical textbook [123], AI for Requirements Problem Solving in this book falls in the so-called “laws of thought”, or logicist approach. In it, the emphasis is on defining rules for how to draw a correct conclusion from some

given information. The rules to apply to the given information when drawing conclusions can reflect the thinking patterns of a typical individual or of an expert.

It is consequently true that AI for Requirements Problem Solving, as it is developed in this book, also suffers from two main limitations of the logicist approach, which Russell and Norvig summarise as follows [123]:

> *"First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem 'in principle' and doing so in practice. Even problems with just a few dozen facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning to try first. Although both of these obstacles apply to any attempt to build computational reasoning systems, they appearerd first in the logicist tradition."*

This book is intended to help solve the first issue, specifically for Requirements Problem Solving. If you yourself have that informal knowledge relevant for solving recurrent Requirements Problem Solving tasks, or you have access to someone who does, the book will suggest how to go from that informal knowledge to languages and algorithms.

As for the second issue, I have no better solution than to map algorithms used for Requirements Problem Solving to well known algorithms in AI. This book will illustrate, for example, that there is quite a lot you can automate in Requirements Problem Solving by using a language whose sentences can be converted into graphs, and by applying algorithms for reasoning on graphs (such as searching sub-graphs with some interesting properties). This will be clearer later in the book.

Chapter 3

Problem and Solution Concepts in Requirements Engineering

Requirements Engineering designates both the practice of rigorously doing Requirements Problem Solving, and the field of research which studies this practice and ways to improve it. This Chapter connects the ideas discussed in Chapters 1 and 2 to the basic ideas and terminology of Requirements Engineering. This is important, because various Requirements Modelling Languages and algorithms, that is, AI for Requirements Problem Solving, have been proposed in Requirements Engineering since the origins of the field in the 1970s.

3.1 Requirements Engineering

This Section suggests the following relationship between Requirements Problem Solving and Requirements Engineering: The former designates the phenomenon which the latter studies and aims to influence.

Requirements Engineering is a term which designates both an engineering discipline and a field of scientific research.

The engineering discipline covers the various activities, such as elicitation, modelling, analysis, negotiation, and so on, which are done in order to define rules that a system has to satisfy when it is made and used. The rules can originate in expectations of stakeholders who invest, use, or otherwise influence and are influenced by the system, in the conditions of the system's operating environment, its regulatory environment, and so on.

It is an engineering discipline. These activities have to be done rigorously, in planned steps, using tried and tested mathematical or other tools.

Requirements Engineering as a field of scientific research studies a variety of topics, such as information elicitation [56, 71, 38], categorization [36, 155, 88], vagueness and ambiguity [108, 99, 85], prioritization [91, 11, 70], negotiation [98, 13, 82], responsibility allocation [36, 23, 51], cost estimation [14, 17, 131], conflicts and inconsistency [110, 69, 145], comparison [108, 99, 100], satisfaction evaluation [16, 108, 93], operationalization [54, 51, 47], traceability [57, 118, 31], and change [26, 149, 20]. Each topic is related to issues and tasks which occur during Requirements Problem Solving.

Historical origins of Requirements Engineering are in software engineering, and specifically in the challenge to define and document what the system should do for its stakeholders and in its environment, without saying exactly how it will do this. The "how" usually remains outside the scope of Requirements Engineering, and is the responsibility of those engineering, making, maintaining, and changing the system.

The notion of *system-to-be*, a term which emphasises that it is not made yet, or simply *system* is central to Requirements Engineering.

Due to the historical origins, system usually designates *software and hardware*. At its boundary are the people who interact with it, and any other things in the environment which the system can somehow exchange information with, influence, and be influenced by.

The phenomenon which the Requirements Engineering discipline and field focus on existed before either the discipline or the field were formally recognised. Requirements Engineering arose in response to situations observed in systems engineering in general, of not knowing how to make sure that the system being made will in fact appropriately address the issues which motivated making it in the first place.

Requirements Problem Solving designates that phenomenon, namely all that people do, when they have unclear, abstract, incomplete, potentially conflicting information about expectations of various stakeholders, and about the environment in which these expectations should be met, a system should be made to satisfy these expectations, and they want to define rules, such that if the system is made to satisfy these rules, then it will also satisfy the expectations in its given environment.

Requirements Problem Solving is present when designing new and changing existing systems. It needs to be done for any 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 applications ecosystems, global supply chains, medical processes, business processes, mobile gaming, and so on. Requirements Problem Solving is done regardless of how the software in the system is designed and made, from a military waterfall approach to a startup's own agile dialect, and from organisations where software engineers talk directly to customers, to those where product designers, salespeople, or others mediate between requirements and code. In all these cases, there will be unsatisfied expectations, and the need to make systems to satisfy them.

3.2 Problem and Solution

This Section introduces definitions for the terms problem and solution, and relates them to the notions of Problem Situation, Solution Situation, and Requirements Problem Solving.

Problem and solution are common terms. The dictionary definition of problem is that it designates "a matter or situation regarded as unwelcome or harmful and needing to be dealt with and overcome".

The corresponding definition for solution is that is is "a means of solving a problem or dealing with a difficult situation".[1]

This Section introduces definitions for "problem" and "solution" which are specific to this book. They are simple, uncontroversial, and coherent with their dictionary definitions. The main benefit of having specific definitions is that they use the terminology introduced for Requirements Problem Solving in Chapter 1. Secondary benefits are less obvious, and I will highlight them below.

3.2.1 Problem

I use the term "problem" to refer to *ideas* about what is observed, or believed to be true in a Problem Situation. Problem is what you observed or think is true in the Problem Situation. It is *not* a record of these ideas, such as, for example, a document where you wrote them down. It is the ideas or thoughts themselves.

It is important that problem designates ideas, not their representations. This is because I argued earlier, in Section 1.2, that different people can see different problems in the same situation. They may pay attention to different events, things, and individuals. They may draw different conclusions about what is and is not desirable in that situation. You may hold one set of ideas, but there is no reason others should share them.

Instead of using the terms "ideas" and "thoughts" in my definitions, it is more conventional in Requirements Engineering to talk of *propositions*. I consequently say that you and I may believe different *propositions* to be true of a situation, even if we are in that same situation. The term "proposition" has a specific definition in philosophy, and I follow the one from Matthew McGrath [104] in the Stanford Encyclopaedia of Philosophy:

> *"Propositions [...] are the 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."*

[1] Both quotations come from a Google search for keywords "define:problem" and "define:solution".

Tying the above to Problem Situation leads me to the following simple definition for the term "problem".

Definition 3.2.1. *Problem:* propositions believed to be true of a Problem Situation. Problem

3.2.2 Solution

Comments I made for the term "problem" apply for the term "solution". Solution are ideas believed to be true of the Solution Situation. Hence the following definition.

Definition 3.2.2. *Solution:* propositions believed to be true of a Solution Situation. Solution

The major difference from the common sense definition of the term "solution" is that here, "solution" is not that which brings about the Solution Situation. It amounts to propositions about the Solution Situation. As I explain in Section 3.3, I use the term system for that which brings about the Solution Situation.

3.3 System

This Section introduces the term system and relates it to the terms introduced so far.

Although the historical origins of Requirements Engineering are in software engineering, the term "system" in contemporary Requirements Engineering is not restricted only software and, or hardware. Its scope can include only limited to specific (parts of) software and hardware, or widened to include such issues as work guidelines, business processes, responsibilities, contracts, or other concerns.

As various things can be part of a system, I prefer not to define the term by saying what can be in it, or has to stay outside. It makes no difference in this book what exactly is, or is not part of a system. What matters is that the system is all that is made and used to bring about a solution.

Definition 3.3.1. *System:* that which is made and used in order to make Solution true. System

A system need not be about software or hardware. It can be a brand, a political election programme, a corporate strategy, or a business process.

In all cases I discuss in this book, the system is not restricted to software and hardware. In some of the cases, software and hardware were not mentioned at all as important parts of systems which were actually used.

3.4 Models

This Section introduces the terms model, Problem Model, Solution Model, and relates them to those introduced so far.

The remaining piece of the puzzle is to describe solutions, problems, and systems in such a way that we can communicate about them during Requirements Problem Solving. This is done with models.

Definition 3.4.1. *Model:* representation of propositions. Model

This is not a conventional use of the term model in Requirements Engineering. Model is normally used to designate the representation of the system only. However, I need to talk about representations of solutions, problems, and systems, which leads me to several kinds of models.

Definition 3.4.2. *Problem Model:* representation of a Problem. Problem Model

Definition 3.4.3. *Solution Model:* representation of a Solution. Solution Model

Definition 3.4.4. *System Model:* representation of propositions believed to be true of the system. System Model

It is on the basis of System Model that the system is implemented, updated, changed, its new releases planned, made, announced, rolled out. The System Model'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.

System Model can take different forms, from minimalistic to-do lists that hint at stakeholders' expectations and subsume implicit design and 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 pseudo-code.

A system is not the output that Requirements Engineering produces. Requirements Engineering does not include, for example, the

detailed engineering, development, testing, release, maintenance, and so on, of software which may be part of the system, nor can it include the training of people who should use it. In the law firm owner's case, the Solution included changes in contracts with employees, in incentives, training, team building, among others. They are activities which, to be done well, each require specific expertise, and are delegated to those who have it.

Problem Models, Solution Models, and System Models are the output sought in Requirements Engineering and Requirements Problem Solving.

3.5 Default Problem and Solution

This Section presents and discusses the Default Problem and Default Solution concepts in Requirements Engineering.

There is a default definition of the Problems that Requirements Engineering tries to solve when applied. There is also a default definition of the Solution sought. It is important to know them, because they highlight a number of assumptions made in Requirements Engineering.

The *de facto* default view in Requirements Engineering is that Requirements Problem Solving is done 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 System Model [15, 36, 59, 110, 49, 155, 144, 23, 122, 83, 47].

This general view of the problem solving process, that you start with less detailed and somehow deficient information, and increase detail and remove deficiencies, is also shared in other domains involving design, such as architecture [137, 95] and civil engineering [4].

Within that view, which Requirements Engineering has of Requirements Problem Solving, what is the default definition of Problems and Solutions?

The most influential treatment of this question is in Pamela Zave and Michael Jackson's seminal paper "Four dark corners of requirements engineering" [155], and is echoed in discussions on the philosophy of engineering [135]. Their view is aligned with some of the most influential research in Requirements Engineering, which

both preceded and followed the said paper. This includes, for example, contributions from Boehm et al. [15, 13], van Lamsweerde et al. [36, 37, 145, 146, 144, 99], Mylopoulos et al. [108, 59, 23], Robinson et al. [122], Nuseibeh et al. [110, 76], to name some.

According to Zave and Jackson Requirements Engineering is successfully completed in any concrete engineering project when the following conditions are satisfied [155]:

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."*

Using the terms I introduced so far, Zave & Jackson's conditions translate as follows:

1. There is a Requirements Model, call it R, which stakeholders agreed on. It represents propositions that convey stakeholders' expectations.
2. There is an Environment Model, called K, which stakeholders agreed on. It represents propositions believed to be true of the environment in which the System will run.

3. There is a System Model, call it S, which describes propositions true of the System.

4. If the propositions represented in the Environment Model are true, that is, the environment is as described, and the System is made and runs in that environment according to the System Model, then propositions described in the Requirements Model will also be true.

5. If the System is made and runs according to System Model, then the environment will remain as described in the Environment Model, and if the environment remains as described, then the System will continue to run without violating System Model.

The translation emphasises that there are *representations* of three kinds of propositions, namely requirements, domain knowledge, and system propositions. The translation also does *not* assume that any of these representations is written in classical logic, and therefore, cannot talk about proofs. Instead, it rewrites the fourth and fifth conditions without assuming the language used to make the representations. All these are minor changes, and the translation preserves the central ideas.

Perhaps the most important observation to make about the conditions from Zave & Jackson is that they do not talk about the structuring of the Problem and Solution, and about the design of the System. In other words, there is no indication that this is ill-structured problem solving. The conditions should be checked after the requirements, environment, and system are clear enough, to make problem solving well-structured.

Returning to the main topic of this Section, the translation suggests a default Problem and Solution for Requirements Engineering.

Default Problem

Definition 3.5.1. *Default Problem:* there are

1. a set R^p of requirements propositions, which are propositions believed to be true of what stakeholders expect, and

2. a set K^p of environment propositions, which are propositions believed to be true about the environment in which the system will be used,

and it is not sufficient for the environment propositions alone to be true, in order for requirements propositions to be true.

The Default Problem is that you know something about stakeholders' expectations and about the environment in which they need to be satisfied, yet that environment alone does not ensure that these expectations indeed are satisfied.

Convention 3.5.2. I write X^p for a set of propositions, and X for the set of representations of propositions, which may, but need not be related to those in X^p. The reason I dissociate X from X^p, is that it is hard to be sure that all propositions in R^p are accurately represented by the content of R. Keep in mind that propositions in R^p are ideas, not representations of ideas. They are hard to access, so to speak, because by being ideas, they are "in the mind" of your own, and of others. Going from R^p to R is complicated, involves having people communicate with you during elicitation about propositions, and thus probably means that you and someone else would produce different R sets, from presumably the same R^p.

In contrast to the Default Problem, the Solution is not the input, but the output of problem solving, comes therefore after problem structuring, and is about Models.

Definition 3.5.3. *Default Solution:* there are

Default Solution

1. a Requirements Model R, which stakeholders agreed on, and which may represent propositions from R^p in the Default Problem,

2. an Environment Model K, which stakeholders agreed on, and which may represent propositions form K^p in the Default Problem,

3. a System Model, call it S, which describes propositions true of the System,

and the System Model S is such that

1. if the propositions represented in K are true, that is, the environment is as described, and the System is made and runs in that environment according to the System Model, then propositions represented in the Requirements Model R will also be true, and

2. if the System is made and runs according to S, then the environment will remain as described in the Environment Model

> K, and if the environment remains as described in K, then the System will continue to run without violating the propositions represented in S.

Keep in mind that K in the Default Solution does not need to represent exactly, all, or any of the propositions in K^p in the Default Problem. Same applies to requirements propositions in R^p and the Requirements Model R. This is because the Default Problem triggers Requirements Problem Solving, which involves problem structuring, and the information known for the original Problem can be removed or replaced.

Chapter 4

Introduction to Requirements Modelling Languages

Requirements Modelling Languages are formal languages specialised for use in Requirements Problem Solving. This Chapter clarifies what a formal language is normally made of, and explains its role in problem solving in general. To give a clearer idea of where Requirements Modelling Languages come from and look like, this Chapter gives a rough historical overview of their design, discusses their broad similarities and differences, and presents two Requirements Modelling Languages called i-star and Techne.

4.1 Formal Language

This Section explains what a formal language is normally made of, regardless of the kinds of problems it is used for. The basic components, syntax and semantics, are explained and a trivial example of a formal language is given.

Formal languages are used for communication, same as natural languages such as English or French. An important difference between a formal language and a natural language, is that every sentence in a formal language is made according to clearly defined and finite set of rules. A definition of a formal language amounts to a number of rules, which together define what sentences of that language can look like and what they are about, that is, what they refer to.

I follow David Harel and Bernhard Rumpe [68] in seeing a formal language as made of two basic sets of rules. They are syntax rules and semantics rules, or simply, syntax and semantics. Each set can be further broken down into pieces, and this Section clarifies the purpose of these pieces, and how they fit together to define a formal language.

4.1.1 Syntax

Syntax rules say which symbols can appear in sentences, and how these symbols are combined into sentences. Syntax is defined with two sets of rules. *Symbol rules* define all allowed symbols. *Grammar rules* define all allowed combinations of symbols.

For illustration, suppose that you want the language L to have sentences in which there are only Arabic numerals symbols. You first define the set of allowed symbols, call it S, as follows

$$S = \{0,1,2,3,4,5,6,7,8,9\}.$$

The above is a set of symbols, not of numbers. I still have not said what each symbol represents, or stands for, and I will do it later with semantics.

Suppose, then, that you want all sentences of this language to include exactly 10 symbols. In other words, any sentence in L has the following format:

␣␣␣␣␣␣␣␣␣␣

where each ␣ must be replaced by any one symbol in the set S. It follows that 0019200216 is a sentence in L, but 108141 is not, and neither is $401-8208w28085<k\%0258$.

To define that all sentences in L must include exactly 10 symbols from S, and that there can be no white spaces between them, you can write that every sentence of L, call that generic sentence a must have the following format:

$$a ::= xxxxxxxxx$$

where x is any member of S, that is $x \in S$. The rule above is itself written in a formal language, called Backus-Naur Form.

If you need no more symbols and no more rules for how to combine symbols, then the syntax of L is defined with the following rules, where the first three are symbol rules, and the fourth is a grammar rule.

$$\begin{aligned}
&L = \{a_1, a_2, \ldots\},\\
&S = \{0,1,2,3,4,5,6,7,8,9\},\\
&x \in S,\\
&a ::= xxxxxxxxx.
\end{aligned}$$

It is important to keep in mind that symbols can be anything really, and it is up to you to choose them and the grammar for combining them into sentences. They do not need to look like anything in natural language. Modern musical notation is one example. You may want to use pictures as symbols, or sketches, or physical objects. As long as they serve the purpose of the language you are making.

4.1.2 Semantics

Semantics are rules which define what the symbols and sentences are meant to represent, and how to determine which symbol and sentence represents what. In more technical terms, semantics defines the *semantic domain* of the language and the *semantic mapping*, which ties sentences to elements in the semantic domain.

Suppose that I want to use the syntax defined above to represent cell phone numbers in Belgium. The semantic domain is therefore the set of all possible cell phone numbers in Belgium.

I know that any cellphone number in Belgium must have ten digits, so sentences in syntax have good length. To map these sentence

to cell phone numbers, it is enough to map each symbol to a number, and I can do this with the following rules:

symbol 0 represents number 0,
symbol 1 represents number 1,
symbol 2 represents number 2,
...,
symbol 9 represents number 9.

The above is the same as defining a function, called the semantic mapping function, or simply semantic mapping. Denote that function m, and let it take any symbol, and return the corresponding number, so that $m(\text{symbol } 1) = 1$, and so on.

I also know that Belgian cell phone numbers always begin with a zero. This is something that I know about the domain, and the language should reflect this. Above, the syntax of L allows any number symbol to be on the first place in the sentence. I therefore change the syntax by using the following rule instead of the previous rule for a:

$$a ::= 0xxxxxxx.$$

The reason why it is relevant to map sentences to numbers, is because of how cell phone operators defined cell phone numbers, and their systems recognise cell phone numbers as 10-digit integer numbers.

Suppose, instead, that a cell phone numbers are not numbers, but that to call someone's cell phone, you have to input ten Latin alphabet letters. In that case, the semantic domain would be all 10-place words made from Latin alphabet letters. The syntax of L would no longer be appropriate, since it has no unique symbol for every smallest relevant element of the semantic domain. This does not mean that Arabic numerals symbols are not good, but only that I can no longer map individual symbols to individual elements of the domain, and ensure that each element in the domain has a corresponding unique representation in syntax. This change in the semantic domain would result in changes to both syntax and semantic mapping.

Defining formal languages is usually harder than in the example above, but this is a fair start. The rest of the book will raise and

discuss many language design issues. The terminology of syntax and semantics will come back every time, and it will keep being used in the same way as above.

4.2 Role in Problem Solving

This Section argues that the role Requirements Modelling Languages have in Requirements Problem Solving and Requirements Engineering is based on the assumption that they influence how Requirements Problem Solving is done, by influencing how information used during problem solving is represented, and that this in turn influences how human problem solvers think during problem solving.

Requirements Modelling Languages are formal languages. They differ from various other types of formal languages in that their syntax and semantics are designed to support specific tasks in Requirements Problem Solving.

It is an implicit assumption in Requirements Engineering that how information is represented during problem solving influences how problem solving is done. That assumption is an important motivation for doing research on and teaching formal languages for Requirements Problem Solving and Requirements Engineering, and on the creation of guidelines, processes, methods for making and manipulating the resulting representations.

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 [92], which is that "[s]tructural differences between language systems will, in general, be paralleled by nonlinguistic cognitive differences" and that "[t]he structure of anyone's 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 [63, 52], which is that [112] "use of the linguistic system [...] actually forces the speaker to make computations he or she might otherwise not make."[1]

[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 [55]: "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

In cognitive science, there are empirical results [157, 34, 80] supporting the claim that "external representations", or sentences of a formal language, are relevant when solving complex problems. They are not only memory aids, but 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 [78].

4.3 Rough Historical Overview

This Section gives a rough historical overview of Requirements Modelling Languages in Requirements Engineering research. The Section uses specialised terminology of Requirements Engineering to mention main similarities and differences between these languages.

Requirements Modelling Languages are formal languages proposed in Requirements Engineering research to support various tasks in Requirements Problem Solving. They arose in response to three intertwined questions which remain among the central ones in the Requirements Engineering research field, and for at least four decades now [61]:

1. What information should be elicited from the stakeholders of the System?
2. How to represent, create models of the elicited information?
3. What kinds of computations should be performed over these models, and why?

The initial response to these questions was to apply formal methods [150, 30] in Requirements Problem Solving. Formal methods are highly developed formal languages for the specification of the properties of a System. Examples of formal methods are VDM [8], Larch [65], Z [133], B [1], Alloy [79].

Since the 1990s, it is recognised in Requirements Engineering research that formal methods are not relevant as formal languages

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."

for Requirements Problem Solving [59]. The main complaint is that they are too generic. They give no indications about which types of information to elicit or represent about the Problem Situation and Solution Situation, how to organise and represent this information in ways which can help such tasks as the negotiation and validation of requirements by stakeholders. Their syntax and semantics are not designed with Requirements Problem Solving in mind. They have no specific features for answering recurring questions in Requirements Problem Solving.

Contemporary view is that Requirements Modelling Languages have the difficult task to bridge the messy steps in Requirements Problem Solving and those when rigorous application of formal methods becomes feasible. This view is reflected in many Requirements Modelling Languages, as they include concepts and rules for how to translate their models, or some parts thereof, into models in formal methods.

The first formal language that was considered a Requirements Modelling Language is RMF [60]. It is "a notation for requirements modeling which combines object-orientation and organization, with an assertional sublanguage used to specify constraints and deductive rules" [59].

An idea in RMF, which remains important still today, is that a Requirements Modelling Language should explain how its sentences can be translated into sentences of a formal logic, such as classical first order logic, or a specialised variant, such as linear temporal first order logic. This ensures that algorithms which can be applied to sentences of such logics can be applied, after translation, to sentences in a Requirements Modelling Language. It also means that representations made with the Requirements Modelling Language can be translated into those of a formal method, as long as the formal method has a translation to the same formal logic as the Requirements Modelling Language.

The semantic domain of RMF distinguished between propositions about entities and about activities. This may be generic and consequently versatile, but it is also still generic. For example, it may be relevant to know which of some given activities are more important than others, or which are more desirable than others, as this may influence deciding which activities to include and exclude from a Solution.

The distinction between entities and activities in the semantic

domain of RMF was judged limited [59] and responses to limitations went in two directions.

One direction, in Requirements Modelling Languages such as KAOS and i-star, is to have more categories and relations predefined in the semantic domain, and to keep that set of categories and relations fixed. This means, for example, that if the language has no category or relation that you can use to indicate that some situation is more desirable than another one, or that an activity is more important, then you cannot show this in models made with this language. The language does not have rules for how to add new categories and relations to it, so that you can, for example, add a new relation and use it to show that a situation shown in a model is more desirable than another situation shown in the same model. To compensate for this, the languages in this approach often include many categories and relations judged relevant for Requirements Problem Solving in general.

The other direction was adopted in the formal language Telos [107] and consists of leaving the concepts and relations in the semantic domain undefined. The language includes tools to define the categories and relations in the semantic domain. This second approach is more versatile, but its abstraction makes it difficult to provide methodological guidance which can be given when a fixed set of concepts is known and manipulated every time the language is used. Although Telos could be used for Requirements Problem Solving, it has rarely had that role.

KAOS remains an important Requirements Modelling Language. "The overall approach taken in KAOS has three components: (i) a conceptual model for acquiring and structuring requirements models, with an associated acquisition language, (ii) a set of strategies for elaborating requirements models in this framework, and (iii) an automated assistant to provide guidance in the acquisition process according to such strategies" [36]. The conceptual model of KAOS defines the categories and relations which make up the semantic domain. There are objects, operations, agents, goals, obstacles, requisites, scenarios, and relations such as specialisation, refinement, conflict, operationalisation, concern, and so on. There are rules in KAOS for how to translate its models into sentences of linear temporal logic.

KAOS introduced many important new ideas in the design of Requirements Modelling Languages. It introduced the concept of

"goal" to Requirements Modelling Languages, and used it to represent stakeholders' expectations, which the system and stakeholders should work together to achieve. KAOS gave the template for the definition of new Requirements Modelling Languages, by showing how to closely fit the language and the guidelines for using it. The language also allowed one to have models which are written only in a visual notation and simple templates, and rewrite parts of models in linear temporal logic only if that was relevant in the specific case of Requirements Problem Solving. This role of linear temporal logic in KAOS made it possible to see KAOS as a language which comes before and naturally precedes the application of formal methods, at least those which also use linear temporal logic. One could use KAOS for Requirements Problem Solving, and once the Solution is found, take relevant parts of the KAOS model of that Solution, and carry them over into a detailed System model made with linear temporal logic.

i-star is language that distinguishes itself from those mentioned above both in its design and its focus. In terms of design, its initial definition did not come with rules for how to translate its models in sentences of a formal logic or formal method. The focus of i-star is on helping the engineers and stakeholders understand the interdependencies of actors within and in relation to a System, their individual and joint goals, tasks, and available or necessary resources, the roles they occupy.

A model in i-star aims to be a snapshot of the intentional states (what they want, assume, know, and so on) of actors in a situation, along with what roles they adopt, and how they depend on each other for the satisfaction of individual and joint goals, the performance of tasks, and use of resources. The System or its components are considered as actors, alongside human individuals and groups.

i-star is a lightweight language relative to KAOS. This should make it easier to learn. This was recognised as a critical feature, given that requirements must be validated by stakeholders who cannot be expected to manipulate artefacts produced with formal methods and formal logics.

i-star was used as the Requirements Modelling Language component of Tropos [23], a methodology for information systems engineering. Once i-star models of the System within its organizational environment are made, Tropos suggests how to proceed towards models of data and behaviour, useful for detailed engineering of

the System. Formal Tropos [51] continued the tradition of mapping models to first order logic sentences, by mapping i-star models to sentences of linear temporal logic. This connected i-star to formal methods, in an analogous way to how KAOS is related to formal methods.

Techne[83] was a more recent development in Requirements Engineering research. It highlighted and promoted three ideas for the design of Requirements Modelling Languages. One is that each Requirements Modelling Language comes with its own assumptions about what Requirements Problem Solving involves, and what the Problems and Solutions should look like. However, it might be the first language which explicitly defined the generic Problems it is used to solve, and gave formal properties that a model has to satisfy, in order to include a Solution to a given Problem. The second idea is that there can be different candidate Solutions to the same Problem, and that it is important for a language to have tools to indicate which of these Solutions is more desirable than others, and to compute which of them is the most desirable, or roughly speaking, "the best". The third idea is that it is possible to have a language with a relatively simple semantic domain, and be able to represent in its models many things that were normally thought to require more categories and relations in the semantic domain of a language.

Another interesting characteristic of Techne is that it is perceived as an "abstract" Requirements Modelling Language intended to be used as the starting point for the definition of new Requirements Modelling Languages. Languages made from Techne are bound to be quite different from RMF, KAOS, and i-star. Techne is not object-oriented and does not incorporate the specialisation relation. Elements of the semantic domain in Techne are propositions. Techne supports neither the definition of temporal constraints, nor task sequencing, nor can it distinguish between domain assumptions which are facts (say, laws of nature) from those which are open to debate. Emphasis is on straightforward knowledge representation and its use towards the identification of candidate solutions.

Techne and i-star, for example, differ in several respects. i-star cannot represent conflict, preferences, or mandatory and optional requirements. Alternative decompositions of a goal in i-star are compared in terms of their contributions to softgoals. Techne keeps softgoals, but due to the vagueness of softgoal instances [86, 88] it requires that they are approximated, meaning "refined" by other non-

softgoals, among which preference relations can be added to indicate which satisfy the softgoal in more desirable ways than others. Techne includes no concepts pertaining to actors and roles.

Paolo Giorgini *et al.* [54] recognised the need to formalise representations of goals identified during Requirements Problem Solving. The aim is to evaluate which goals will be satisfied, and how much. Their goal models are AND/OR graphs, in which nodes are goals, and a number of relations is provided to indicate if the interaction is positive or negative (how the satisfaction of a goal influences the satisfaction of the other goal related to it), as well as to specify the strength of the interaction. Techne uses preferences to indicate in the relative degrees of satisfaction, while quantitative estimates of satisfaction levels are not used.

Techne's handling of inconsistency is similar in aim to Anthony Hunter and Bashar Nuseibeh's in LQCL. They are interested in reasoning on an inconsistent models and "keeping track of deductions made during reasoning, and deciding what actions to perform in the presence of inconsistencies" [76, pp.363–364], while avoiding drawing trivial conclusions from inconsistent models. A Techne model keeps track of all the deductions made, and inconsistencies (conflicts) are clearly shown. A significant difference is that their work is based on clausal resolution, which may lead to concluding inconsistency, but this is prevented from leading to irrelevant formulas being inferred. In contrast, Techne addresses directly the identification of maximally consistent sub-models, from which inconsistency cannot be concluded.

Techne by its very design avoids asking stakeholders for quantitative estimates of preference, in contrast to, for example, the language from Sotirios Liaskos *et al.* [100]. Preferences are binary relations, and two preferences cannot be compared in a Techne model itself, but only after the comparison table is constructed. Techne thereby recognizes that there are different approaches to decision-making in the presence of multiple criteria and no ideal decision rules, leaving it to the designer who makes a new Requirements Modelling Language from Techne to choose herself the decision rules to apply on the comparison table.

This Section inevitably leads to the conclusion that Requirements Modelling Languages come in different shapes and forms. In a summary, RMF is a custom formal language with built-in abstraction mechanisms, including aggregation, classification, and generalisa-

tion. KAOS uses the language of 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 Requirements Engineering. i-star has a custom visual notation, which comes together with axioms which instruct how to make and read i-star 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.

Despite the important position that formal languages play in Requirements Engineering, there are no widely-accepted and precise standards that a formal language must satisfy in order to be called Requirements Modelling Language. The evolution of formal languages in Requirements Engineering is one of testing of and converging on similar ideas, rather than making languages according to strict rules of what makes a Requirements Modelling Language.

4.4 i-star

This Section outlines the Requirements Modelling Language called i-star. A simple example is used to illustrate the language.

This Section gives a brief overview of i-star. The aim is not to give a detailed definition, but only give an idea of what the language looks like, the kinds of models that can be made with it, and of what makes its syntax, semantic mapping, and semantic domain. If you are interested in a detailed definition, I suggest looking up Eric Yu's PhD thesis [152].

i-star remains influential since it was proposed in the 1990s. It is the Requirements Modelling Language in the information systems engineering methodology Tropos [23, 51]. There are many extensions of i-star, discussions of its merits and limitations, and 2015 will witness the eight annual i-star research workshop. The state of the art on i-star is discussed in Yu's "Social Modelling for Requirements Engineering" [46].

Unless I indicate otherwise, all citations in this Section are from

Eric Yu's "Modeling organizations for information systems requirements engineering" [153].

4.4.1 Motivation

The aim with i-star was to be able to represent information about how individuals and systems in an organisatonal environment interact, and depend on one another, in their tasks, in using resources, and achieving individual or joint goals. It is a language focused on "organizational environments – an important class of environments within which many computer-based information systems operate" [153].

The emphasis in i-star is on representing how individuals, software, and other resources coordinate to realise personal and joint goals. Every situation one represents with i-star conveys the organisation of agents, and every agent "depends on others for accomplishing some parts of what it wants, and are in turn depended on by others. Agents have wants that are met by others' abilities, run tasks that are performed by others, and deploy resources that are furnished by others. These dependencies form a complex and intricate network of intentional relationships among agents that might be called the intentional structure of the organizational environment".

Using the terms from earlier Chapters in this book, i-star is a formal language which can be used to make Problem Models and Solution Models. There will usually be one i-star model, or set of sentences in the language, to represent the Problem. There will be another to represent the Solution. The first is often called the as-is model, and the second the to-be model.

For illustration, consider a system for scheduling meetings. The person scheduling the meeting, the meeting scheduler, should try to select a convenient date and location, such that most potential participants can participate. Each meeting participant should provide acceptable and unacceptable meeting dates based on a personal agenda. The scheduler will suggest a meeting date that falls in as many sets of acceptable dates as possible, and is not in unacceptable date sets. The potential participants will agree on a meeting date once an acceptable date is suggested by the scheduler.

A model for such a System can be represented as an instance of the i-star Strategic Rationale model or a Strategic Dependency model. The latter kind of models are made with a subset of i-star,

and are used when it is not relevant to represent all information that a Strategic Rationale model would include.

An example Strategic Rationale diagram for the scheduler is reprinted in Figure 4.1 [154]. It shows actors such as *Meeting Scheduler* and *Meeting Participant*, their interdependencies in the achievement of goals, the execution of tasks, and the use of resources, and their internal rationale when participating in the given System. For example, the *Meeting Be Scheduled* goal of the *Meeting Initiator* can be achieved (represented via a so-called means-ends link) by scheduling meetings in a certain way, consisting of (represented via task-decomposition links): obtaining availability dates from participants, finding a suitable date (and time) slot, proposing a meeting date, and obtaining agreement from the participants. Cloud-shaped elements designate so-called softgoals, which differ from goals in that there are no clearly defined and agreed criteria for their satisfaction. Softgoals are commonly used to represent nonfunctional requirements in a goal diagram.

4.4.2 Syntax

The legend in Figure 4.1 gives the symbols that can be used in i-star models. There are shapes labeled “goal”, “softgoal”, “resource”, “task”, “actor”, and those labeled “task-decomposition link”, “means-ends link”, “contribution to softgoal”, and “dependency link”. In any given model, each shape has its own label, and the label is in a natural language, such as English in the Figure.

Rules missing from the legend are those of grammar. They are as follows, for the Strategic Rationale diagram:

- Inside the dashed area of “actor boundary” symbol, there can be any number of i-star symbols, as long as they satisfy all i-star grammar rules.
- “Task-decomposition link” symbol has its source side and its target side, drawn with at short line across the long line. Its source side must be drawn connected only to one “goal”, “task”, or “resource” symbol. Its target side must be drawn connected to one “task” symbol.
- “Means-ends link” symbol has a source and a target side, drawn with an arrow. Its source side must be drawn connected to one

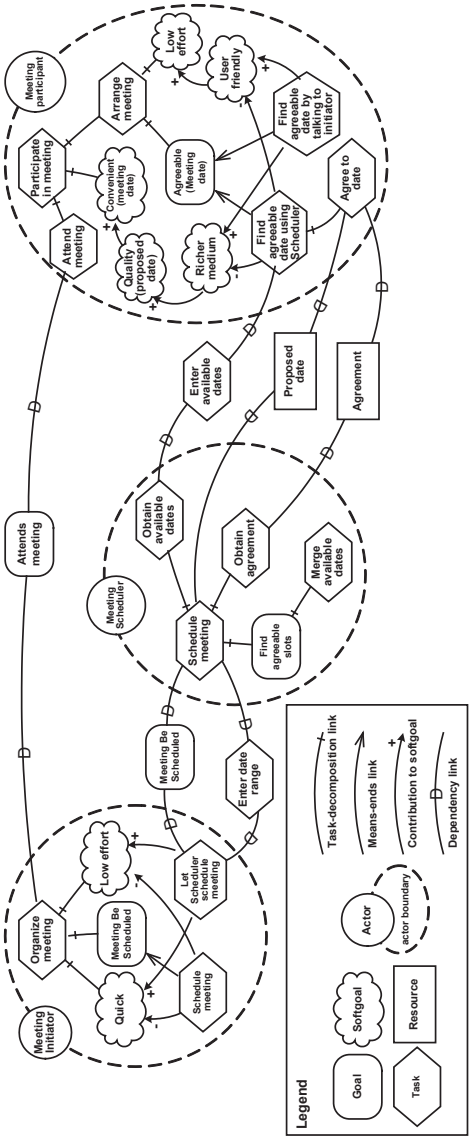


Figure 4.1: An i-star Strategic Rationale diagram from Yu *et al.* [154].

"goal", "task", or "resource" symbol. Its target side must be drawn connected to one "goal" symbol.

- "Contribution to softgoal" symbol has a source and a target side, drawn with an arrow and a label symbol. The label symbol can be exactly one of "+", "++", "-", or "- -". Its source side must be drawn connected to one "goal", "task", "resource", or "softgoal" symbol. Its target side must be drawn connected to one "softgoal" symbol.

- "Dependency link" symbol has a source and a target side, the former on the left-hand side of the symbol "D" and the other on its right-hand side. Its source side must be drawn connected to one "goal", "task", or "resource" symbol. Its target side must be drawn connected to one "goal", "task", or "resource" symbol.

You can draw i-star Strategic Rationale diagrams by applying the grammar rules above to the allowed symbols. There may be other grammar rules, and there are other ways to define them. You can refer to Eric Yu's original work on i-star [153, 152] for the original grammar of i-star.

4.4.3 Semantic Domain and Mapping

The semantic domain of i-star are propositions about goals of agents, how agents depend on one another to achieve goals, tasks that agents do to achieve goals, and resources used when executing tasks and achieving goals. Identifying goals, tasks, resources, dependencies, and so on, is a task for the modeller. She identifies these through observation of the environment and existing systems, during requirements elicitation, among others. To do so, she relies on definitions of notions of goal, task, resource, dependency and others, which are given in the definition of the i-star language, which at the same time define its semantic domain. For illustration, here is how dependencies are defined, or explained:

- Goal Dependency is a relation in which "one agent, the depender, depends on another, the dependee, for the fulfillment of a goal. The dependee is free to choose how to accomplish the goal. The depender is only interested in the outcome".

- In a Task Dependency "a depender agent depends on some dependee agent for the performance of a task. The task specification constrains the choices that the dependee can make regarding how the task is to be carried out."

- Resource Dependency is a relation in which "a depender agent presupposes the availability of a resource, which is made available by a dependee agent".

The semantic mapping is a function which takes such a proposition, and produces a combination of symbols and labels on symbols, which are all drawn in a Strategic Rationale diagram.

For example, this sentence is a representation of a proposition in written English: "In order to organise a meeting, the meeting initiator depends on meeting participants to attend the meeting". The semantic mapping function would take the proposition and produce an "actor" symbol with label "meeting initiator", a "task" symbol with label "organise meeting", have that "task" symbol inside the "actor boundary" dashed line of "actor" symbol labelled "meeting initiator", there would be a "dependency link" symbol with the "organise meeting" symbol at its source and "attend meeting" goal, and so on. The result are symbols on top of Figure 4.1. Clearly, it is hard to automate the application of the semantic mapping function. It is the modeller who does the work of matching symbols and their combinations to the propositions, based on her understanding of the definition of the semantic domain of i-star.

4.4.4 Comments

I will close this Section with my own opinion of i-star, and in relation to the topics of this book. I see it as a language which has an innovative design that makes it perhaps more accessible (relative to other Requirements Modelling Languages).

Its innovation is not so much in its mathematical aspects, that is, in the axioms which define its concepts and relations via such concepts as "belief" and "intention" taken from knowledge representation and reasoning research in AI. A major innovation is in the idea that an organisational environment, in which a future system should run, can be looked at not in terms of concrete tasks and processes only, but in terms of less precise notion of dependency, which do

hint at who needs to do what, and perhaps who needs to do it before someone else, but highlight how collaboration and delegation generate vulnerabilities of the actors involved.

With as-is i-star models, you can show goals, tasks, resources that actors collaborate on and use, and how, because of that collaboration, they are vulnerable to each other (if one fails, perhaps others, who depend on her, will fail in their goals and tasks as well). With to-be i-star models, you can show how the future system would change existing patterns of dependencies, that is, current collaboration and vulnerabilities.

The emphasis on visual syntax, that is, on having models be diagrams (rather than, for example, formulas), and the small set of concepts and relations (relative to, say, KAOS), makes i-star accessible to novices. It is not odd that research on i-star is an active domain.

4.5 Techne

This Section outlines the Requirements Modelling Language called Techne. The Section uses a simple example to illustrate the language.

4.5.1 Motivation

Alexander Borgida, Neil Ernst, John Mylopoulos and I made Techne in response to a new so-called Core Ontology for Requirements Engineering, or CORE [88]. In general, an ontology is a specification of a conceptualisation of a domain, that is, it is a precise definition of the categories of things or ideas in a domain and of relations between these things or ideas.

What I mean by "in response" is that Techne was made so that CORE defines its semantic domain. This was interesting, because John Mylopoulos, Stéphane Faulkner and I presented CORE as a revision of the ontology that Pamela Zave and Micheal Jackson defined, when they proposed the rules for when Requirements Engineering is done in a systems engineering project. I cited these rules in Section 3.5, and used them to define the Default Problem.

When you propose a new ontology such as CORE, which is designed to be a core ontology, meaning, a basis for other ontologies in Requirements Engineering, and if it is different than the existing core ontology used to define the Default Problem and Default Solution,

you have also to give the Problem and Solution concepts which use the concepts and relations of that new core ontology. CORE comes with its own definition of the Problem which reflects what triggers Requirements Problem Solving, and what problem solving needs to produce.

The Problem and Solution in CORE are called the CORE Problem and CORE Solution below.

Definition 4.5.1. *CORE Problem:* there are CORE Problem

1. a set G^p of goal propositions, which are propositions believed to be true of what stakeholders expect and whose truth can be verified without a doubt, and

2. a set Q^p of quality constraints propositions, which are propositions believed to be true of the quality stakeholders expect from the System, and whose truth can be verified without a doubt,d

3. a set U^p of softgoal propositions, which are vague propositions believed to be true of the quality stakeholders expect from the System, but whose truth (because they are vague) cannot be verified without a doubt,

4. a set K^p of environment propositions, which are propositions believed to be true about the environment in which the system will be used,

5. a set M^p of mandatory propositions, which are propositions which are required to be true, and can be goals, quality constraints, or softgoals, that is $M^p \subseteq G^p \cup Q^p \cup U^p$, and

6. a set P^p of preference propositions, which are true of stakeholders' preference for some goals, quality constraints, and softgoals over others,

and it is not sufficient for the environment propositions alone to be true, in order for all mandatory goal and quality constraint propositions to be true.

I will clarify later what is meant by goals, quality constraints, softgoals, and preferences. The CORE Solution is defined as follows.

Definition 4.5.2. *CORE Solution:* there are CORE Solution

1. a Goal Model **G**, which stakeholders agreed on, and which may represent propositions from G^p in the CORE Problem,

2. a Softgoal Model **S**, which stakeholders agreed on, and which may represent propositions from U^p in the CORE Problem,

3. a Quality Constraints Model **Q**, which stakeholders agreed on, which may represent propositions from Q^p in the CORE Problem, and which are said to approximate the mandatory propositions in the Softgoal Model U,

4. an Environment Model **K**, which stakeholders agreed on, and which may represent propositions from K^p in the CORE Problem,

5. a Preference Model **P**, which stakeholders agreed on, and which may represent propositions from P^p,

6. a Task Model **T**, which describes propositions true of what System and other agents in the environment will do,

and the Task Model **T** and the Environment Model **K** are such that

1. if the propositions represented in **K** are true, that is, the environment is as described, and the propositions in **T** are true, that is, tasks are done as described, then all mandatory propositions represented in **G** and **Q** will also be true,

2. because of approximation, all mandatory propositions represented in **S** will also be true,

3. tasks in **T** are feasible, meaning that executing them does not make false the propositions represented in **K**, and

4. none or some of non-mandatory (optional) propositions represented in **G**, **Q**, or **S** are also true.

The last condition in the CORE Solution is due to there being preferences over propositions which are can be false. It follows that you could find different Solutions, and preferences could tell you which of them is "the best". I will return to preferences and the issue of identifying the best Solution in Chapter 14.

CORE and its Problem recognised that in addition to goals (which correspond to requirements in the Default Problem) and tasks (which

correspond to the System Model there), different stakeholders have different preferences over goals or tasks, that they are interested in choosing among alternative Solution s to the Problem, that potentially many such Solutions can be identified, and that requirements are not fixed, but change with new information from the stakeholders or the environment. In absence of preferences, as in RMF, KAOS, and i-star to some extent, it is (i) not clear how Solution Models can be compared, (ii) what criteria (should) serve for comparison, and (iii) how these criteria are represented.

Techne takes CORE as the definition of its semantic domain. This also influences what Problems and Solutions amount to in Techne, and they are strongly inspired, though not exactly the same as the CORE Problem and CORE Solution above. The differences come from the fact that Techne wants to give a formal, mathematical definition of the Problem and Solution, which requires making decisions about how to have mathematics which convey the ideas represented in the various English words and phrases in the definitions above.

As there could be more than one Solution, Techne talks of *Candidate* Solutions, which are compared on the basis of which preferred and, or optional goals, quality constraints, and softgoals the corresponding Solution Models satisfy (make true).

Another distinguishing characteristic of Techne, is that it was designed as a language core on which to build new Requirements Modelling Languages. By core language, it is meant a minimal set of components which are argued as necessary to a Requirements Modelling Language, if it is to model goals, softgoals, quality constraints, domain assumptions, and tasks used to define Problem Models, to define Solution Models, to model preferences and optional requirements, and use them as criteria for the comparison of candidate Solutions represented in Solution Models.

The simplest way to make a Requirements Modelling Language from Techne is to add a visual syntax. For example, add a diagrammatic notation, and map its syntactic elements to those of Techne's syntax.

4.5.2 Semantic Domain and Mapping

I will follow our original presentation of Techne [83] and explain its semantic domain via categories it uses for the classification of problem solving information, and relations it is interested in, over

pieces of this information.

Classification

Elicited information is classified according to the rules in CORE. The overall idea is to distinguish in a statement, which a stakeholder communicates, the psychological mode from the proposition that the statement represents. Then, you establish which CORE concept the statement instantiates, and this based on the psychological mode (belief, desire, and so on) and on some properties of the statement itself. Stakeholder desires become instances of the goal concept, if they refer to conditions, the satisfaction of which is desired, binary and verifiable (for example, "Deliver music to clients via an online audio player"). If desires constrain desired values of non-binary measurable properties of the system-to-be, then they are instances of the quality constraint concept ("The bitrate of music delivered via the online audio player should be at least 128kb/s"). When desired values are vaguely constrained and on not necessarily directly measurable properties, they instantiate the softgoal concept ("Buffering before music starts in the audio player should be short"). Stakeholder intentions to act in specific ways become instances of tasks to be accomplished either by the system-to-be, or in cooperation with it, or by stakeholders themselves. Beliefs are instances of domain assumption, stating conditions within which the system-to-be will be performing tasks in order to achieve the goals, quality constraints, and satisfy as best as feasible the softgoals. Stakeholder evaluations of requirements — their preferences for some goal (or otherwise) to be satisfied (true) rather than another, or that some must be satisfied, while others are optional — result in relations over requirements subsequently used to compare candidate Solutions.

To solve the Problem, it is necessary that the categorized statements are recorded, refined, expanded by iteratively acquiring new ones. To record requirements, Techne maps statements to labels, thereby sorting them. Let p, q, r (indexed or primed as needed) represent propositions, and $\mathbf{g}()$, $\mathbf{q}()$, $\mathbf{s}()$, $\mathbf{t}()$, and $\mathbf{k}()$ be labels for, respectively, goals, quality constraints, softgoals, tasks, and domain assumptions. A labelling function simply follows the rules of CORE recalled above: if p is an instance of goal, then we write $\mathbf{g}(p)$, if q is an instance of quality constraint, we write $\mathbf{q}(q)$, and so on. Hereafter, *requirement* is synonym for any labeled representation of a

proposition.

Relations

There are five relations on requirements in Techne: (i) *inference*, (ii) *conflict*, (iii) *preference*, (iv) *is-mandatory*, and (v) *is-optional* relations. The first two are used to describe and distinguish between candidate Solutions, the last three to compare candidate Solutions.

Inference When a requirement (a goal, quality constraint, softgoal, task, or domain assumption) is the immediate consequence of another set of requirements, the former is called the conclusion, the latter the premises, and they stand related through the inference relation.

Say there are two goals, $\mathbf{g}(r_1)$ and $\mathbf{g}(r_2)$, with r_1 for "Music plays in a player integrated in the web page" and r_2 for "Player has all standard functionalities for listening music". If there is also a domain assumption $\mathbf{k}(\gamma_1)$, with γ_1 for "If r_1 and r_3 then music is delivered to clients via an online audio player", then we can conclude the goal $\mathbf{g}(r_3)$, with r_3 for "Deliver music to clients via an online audio player". From two goals and an assumption stating a conditional, the conclusion is another goal.

Reading this backwards, from $\mathbf{g}(r_3)$ to the three premises, resemblance to refinement becomes clear: the inference relation can be used to connect the refined requirement to the requirements that refine it. The refinement of a goal by other goals has been a salient feature of KAOS, while other Requirements Modelling Languages had their own proxies (for example, task decomposition in i-star) of the refinement relation. The intuitive meaning of these relations is that if the set of more precise requirements is satisfied, then the less precise requirements are assumed satisfied.

Techne considers that, say, goal refinement and task decomposition ask basically the same question: What more precise requirements should be satisfied in order to assume that the less precise — refined, decomposed — requirement is satisfied as well? Instead of relating less precise to more precise requirements by a refinement or decomposition relation, Techne generalizes these via the inference relation. Note that in both these cases the form of the rules $\mathbf{k}(\phi)$ is a *definite Horn clause* [115].

Conflict Contradictory/inconsistent requirements cannot be in the same candidate Solution, or equivalently, are in conflict. The conflict relation stands between all members (two or more) of a minimally inconsistent set of requirements. That a candidate Solution should be conflict-free means that conflict relations play a crucial role in distinguishing between consistent sets of requirements, and if these sets satisfy some additional properties, in distinguishing between candidate Solutions. To say that n requirements are in direct conflict, another piece of information is needed, namely an implication which explicitly states that if these requirements together hold, then they imply an inconsistency: for example, to say that $\mathbf{g}(r_1)$ and $\mathbf{k}(r_4)$ are in conflict, where r_4 is for "The user cannot download the audio files", it is necessary to say that the two are contradictory, which is done via an assumption: for example, $\mathbf{k}(\gamma_2)$, with γ_2 for "$\mathbf{g}(r_1)$ and $\mathbf{k}(r_4)$ are contradictory".

Preference Stakeholder evaluations of requirements convey that not all requirements are equally desirable. For example, perhaps "The bitrate of music delivered via the online audio player should be at least 256kb/s" is strictly preferred to "The bitrate of music delivered via the online audio player should be at least 128kb/s". If a requirement is strictly more desirable than another one, then there is a preference relation between them and by *strictly*, we mean that they cannot be equally desirable.

Is-mandatory Evaluation is not only comparative, as in the case of preference: individual requirements can be qualified in terms of desirability regardless of other requirements. The is-mandatory relation on a requirement indicates that the requirement *must* be satisfied, or equivalently, that a conflict-free set of requirements which does not include that requirement cannot be a candidate Solution. If $\mathbf{k}(r_4)$ is mandatory, then every candidate Solution will include it, and exclude all requirements contradicting $\mathbf{k}(r_4)$ (because a candidate Solution cannot include conflicts).

Is-optional In contrast to the is-mandatory relation, the is-optional relation on a requirement indicates that it would be desirable for a conflict-free set of requirements to include that requirement, but that set can still be a candidate Solution if it fails to include the optional requirement; for example, if $\mathbf{k}(r_4)$ is optional, then a conflict-free set

of requirements which does not contain $\mathbf{k}(r_4)$ can still be a candidate Solution. Stated otherwise, if there are two candidate Solutions which *differ only* in that one has an optional requirement and the other not, then the former is strictly more desirable than the latter.

4.5.3 Syntax and More Semantic Mapping

Requirements and relations between them are recorded in graphs called *r-nets*. Each requirement and each relation obtains its own node in an r-net, while edges are unlabeled and directed, having contextual informal interpretation: how one reads/calls an edge depends on which requirements and relations it connects (see below). Note already that, as an r-net contains all requirements and all relations for a system-to-be: the r-net thus defines the requirements problem for a given system-to-be, and includes all (if any) candidate Solutions to the problem, so that it is by the analysis of the r-net that candidate Solutions are sought (see §4.5.4).

Modelling Inference

To show an inference relation, put in the r-net a node (**inf**) for the inference relation, then a line from every premise requirement node to **inf**, and a line from **inf** to the conclusion requirement node.

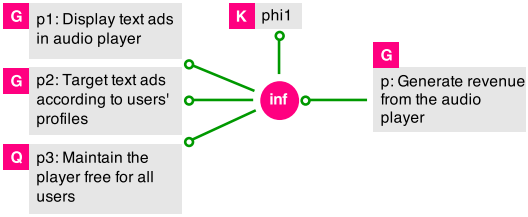


Figure 4.2: Inference as refinement in Example 4.5.3.

Example 4.5.3. Assume the aim is to build a system that would deliver music on-demand: a user visits a website, chooses songs from a database, and can play them in the audio player on the website. Let $\mathbf{g}(p)$, with p for "Generate revenue from the audio player". We can refine it with two goals and a quality constraint: $\mathbf{g}(p_1)$, $\mathbf{g}(p_2)$ and $\mathbf{q}(p_3)$, where p_1 is for "Display text ads in the audio player", p_2 for

"Target text ads according to users' profiles" and p_3 for "Maintain the player free to all users". To conclude $\mathbf{g}(p)$ from $\mathbf{g}(p_1)$, $\mathbf{g}(p_2)$ and $\mathbf{q}(p_3)$, we need to assume that $\mathbf{k}(\phi_1)$, with ϕ_1 for "if $\mathbf{g}(p)$ from $\mathbf{g}(p_1)$, $\mathbf{g}(p_2)$ and $\mathbf{q}(p_3)$, then $\mathbf{g}(p)$". Figure 4.2 shows the r-net with this refinement.

Figure 4.2 shows an r-net in a trivial visual syntax, vaguely inspired by the furniture collection "Un piccolo omaggio a Mondrian" by Ettore Sottsass. The mapping to the symbolic syntax of Techne is obvious from this and other Figures in this Section. My aim was to brighten up the discussion, as it already has its share of dry formulas.

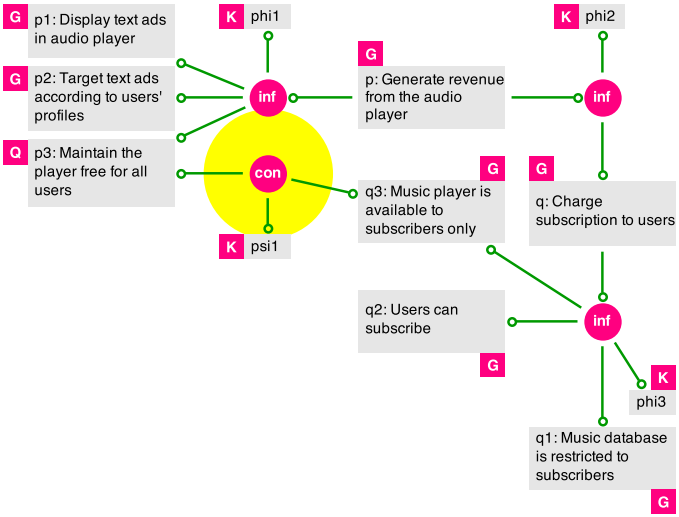


Figure 4.3: Conflict in Example 4.5.4.

Modelling Conflict

To show a conflict between requirements, put a node for each one of the conflicting requirements in the r-net, a conflict node (**con**), and a line from every requirement node in the conflicting set to the conflict node.

Example 4.5.4. (Contd. Example 4.5.3) We start with $\mathbf{g}(q)$ with q for "Charge subscription to users", and add $\mathbf{k}(\phi_2)$, with ϕ_2 for "if $\mathbf{g}(q)$ then $\mathbf{g}(p)$". We then refine $\mathbf{g}(q)$ onto $\mathbf{g}(q_1)$, $\mathbf{g}(q_2)$, and $\mathbf{g}(q_3)$, with q_1 for "Music database is restricted to subscribers", q_2 for "Users can

subscribe" and q_3 for "Music player is available to subscribers only". This requires the assumption $\mathbf{k}(\phi_3)$, ϕ_3 for "If $\mathbf{g}(q_1)$, $\mathbf{g}(q_2)$, and $\mathbf{g}(q_3)$, then $\mathbf{g}(q)$". It appears that "we cannot both maintain the player free to all users ($\mathbf{q}(p_3)$) and make music available to subscribers only ($\mathbf{g}(q_3)$)" which ψ_1 abbreviates, so we add $\mathbf{k}(\psi_1)$. We thereby have the conflict between $\mathbf{q}(p_3)$ and $\mathbf{g}(q_3)$, highlighted in Figure 4.3.

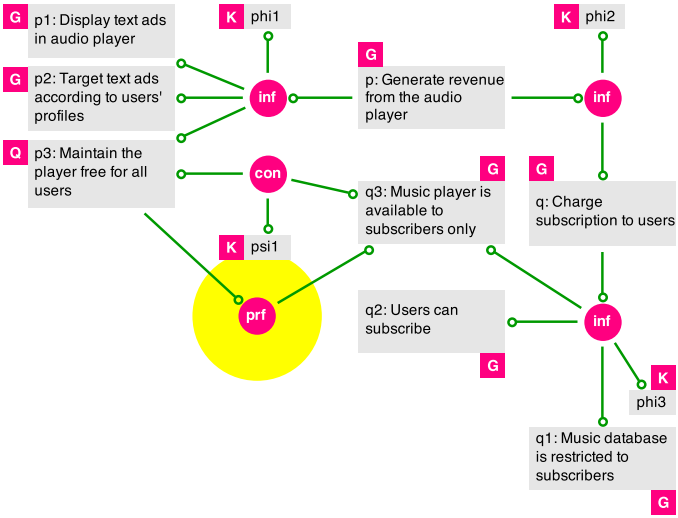


Figure 4.4: Preference in Example 4.5.5.

Modelling Preferences

Preference is a binary relation: if a requirement x is preferred to requirement y, add a preference node (**prf**), and draw a line from the preferred requirement (x) to the preference node (**prf**), and from the preference node (**prf**) to the less preferred requirement (y).

Example 4.5.5. (Contd. Example 4.5.4) The r-net in Figure 4.3 includes two refinements of $\mathbf{g}(p)$. The conflict **con** indicates that these are two *alternative* refinements, as they cannot appear together in a candidate Solution. The preference highlighted in Figure 4.4 says that $\mathbf{g}(q_3)$ is strictly preferred to $\mathbf{q}(p_3)$. This preference becomes one (of potentially many) criteria for the comparison of candidate solutions: if this were the only criterion, then we would choose the

candidate Solution which includes **g**(q_3) instead of another which includes **q**(p_3).

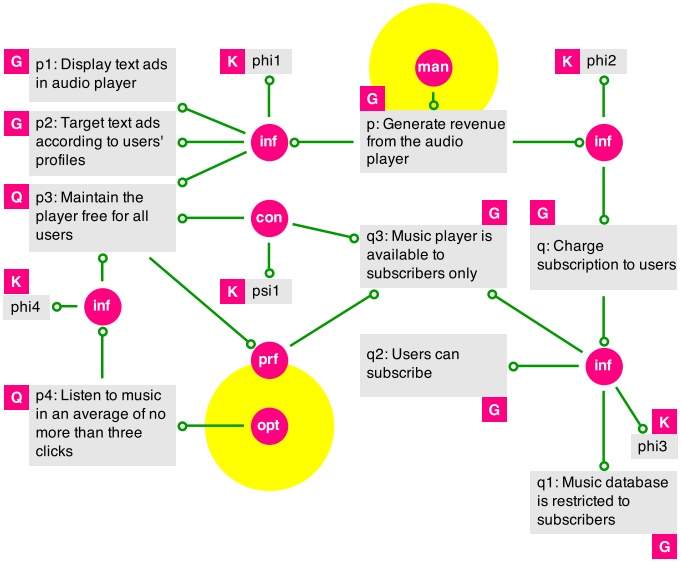


Figure 4.5: Mandatory and optional in Example 4.5.6.

Modelling Mandatory and Optional Relations

Both the is-mandatory and is-optional relations are unary. To say in an r-net that a requirement is mandatory, add a node (**man**) for the is-mandatory relation, and a line from the requirement node to the is-mandatory node. To state instead that a requirement is optional, add a node (**opt**) for the is-optional relation, and a line from the requirement node to the is-optional node.

Example 4.5.6. (Contd. Example 4.5.5) If every Solution must include **g**(p), then we add the node **man** to the r-net in Figure 4.4 and a line from **g**(p) to **man**, as shown in Figure 4.5.

To illustrate the use of the is-optional relation, suppose that maintaining the player free to all users (**q**(p_3)) will allow new users to listen to music in an average of no more than three clicks through the audio service (as they do not need to register or provide their billing

details); we denote $\mathbf{q}(p_4)$ the latter quality constraint. We add $\mathbf{q}(p_4)$ as a node to the r-net, along with the assumption $\mathbf{k}(\phi_4)$, with ϕ_4 for "if $\mathbf{q}(p_3)$, then $\mathbf{q}(p_4)$", and thus an inference relation. Let $\mathbf{q}(p_4)$ be optional: to make it so in the r-net, we connect it to the node **opt**. If we consider the r-net in Figure 4.5, it is no longer obvious which of the two refinements is more desirable than the other: if a candidate Solution includes $\mathbf{g}(q_3)$, then it will not contain $\mathbf{q}(p_4)$, but will have the preferred $\mathbf{g}(q_3)$; if a candidate Solution includes $\mathbf{q}(p_3)$, then it will have $\mathbf{q}(p_4)$, but not the preferred $\mathbf{g}(q_3)$.

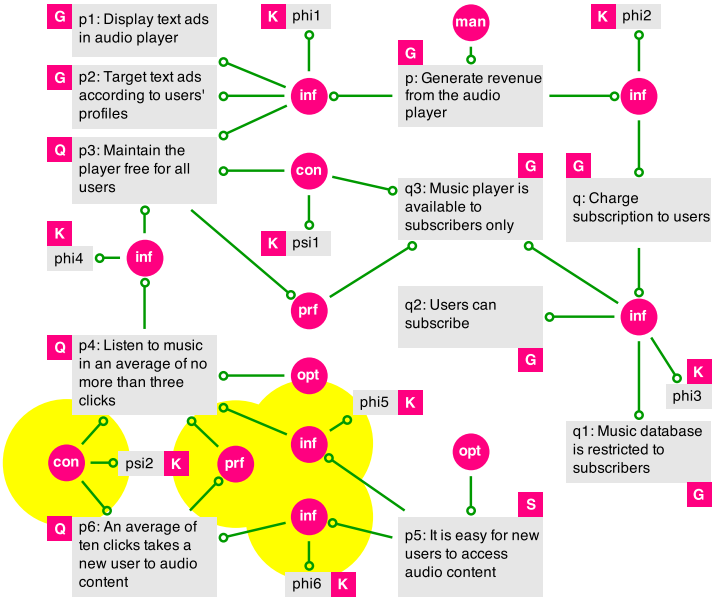


Figure 4.6: Softgoal approximation in Example 4.5.7.

Softgoal Approximation

As softgoals vaguely constrain values of properties that are not necessarily directly measurable, every softgoal ought to be approximated in an r-net. A set of requirements can be an *approximation* of a softgoal if it is assumed that, once its members are satisfied, the softgoal will be satisfied to some extent. As different approximations may

satisfy the same softgoal to different extents, preference relations can be added between the members of different approximations. These preferences let us compare approximations in terms of how well one satisfies the softgoal relative to others.

Example 4.5.7. (Contd. Example 4.5.6) We introduce the optional softgoal $\mathbf{s}(p_5)$, with p_5 for "It is easy for new users to access audio content" into the r-net from Figure 4.5. There are no universal criteria that tell us what "easy" precisely means in the context of this system-to-be. There are thus different ways to approximate $\mathbf{s}(p_5)$. One of them is to say that the smaller the average number of clicks needed to a new user to access audio content (computed over some number of sessions and for a given focus group), the easier it is to access that content. We can introduce at least two quality constraints, one being $\mathbf{q}(p_4)$ and another $\mathbf{q}(p_6)$, with p_6 for "An average of ten clicks are needed to a new user to get to audio content", with corresponding assumptions $\mathbf{k}(\phi_5)$ and $\mathbf{k}(\phi_6)$, with ϕ_5 for "if $\mathbf{q}(p_4)$, then $\mathbf{s}(p_5)$" and ϕ_6 for "if $\mathbf{q}(p_6)$, then $\mathbf{s}(p_5)$". A preference is added, to indicate that the approximation by $\mathbf{q}(p_4)$ is preferred to the approximation by $\mathbf{q}(q_6)$. Finally, we abbreviate "$\mathbf{q}(p_4)$ cannot be satisfied together with $\mathbf{q}(p_6)$" by ψ_2, and add a conflict between $\mathbf{q}(p_4)$ and $\mathbf{q}(p_6)$, along with the assumption $\mathbf{k}(\psi_2)$.

Modelling with Another Visual Syntax

Concrete Requirements Modelling Languages (for example, KAOS, *i**) have a visual syntax as a diagrammatic notation that aims to simplify the making and reading of requirements models created with these languages. Techne is an *abstract* Requirements Modelling Language as it has no visual syntax. To make a concrete Requirements Modelling Language out of Techne, it would be necessary to add a visual syntax, as I did in the Figures in this Section.

4.5.4 Analysis

Analysis in Techne should answer two questions: given an r-net, (i) What are the candidate Solutions to the Problem in it? and (ii) What are the preferred and optional requirements that each candidate Solution contains? Example 4.5.8 informally presents how these answers are sought; we then look into the formalization of the r-nets towards the automation of analysis.

Example 4.5.8. (Contd. Example 4.5.7) A candidate Solution must be conflict-free, so that we are interested in conflict-free subnets of the r-net in Figure 4.6. There are many conflict-free subnets in Figure 4.6: for example, $\mathbf{g}(p)$ taken alone is a conflict-free subnet, as is the refinement shown in Figure 4.2. Since $\mathbf{g}(p)$ is itself a subnet of the said refinement, we are more interested in the entire refinement than in any one of its subnets alone. Stated otherwise, conflict-free (consistent) subnets can be ordered by the subset relation $\subseteq$, and instead of looking for all consistent subnets, those maximal with regards to $\subseteq$ are the most interesting ones. Figures 4.7 and 4.8 highlight two maximal consistent subnets in the r-net from Figure 4.6. These are, however, *not* also candidate Solutions to the requirements problem, as each has goals and quality constraints as source nodes (nodes without incoming lines). Recall that we are interested in finding tasks and domain assumptions which satisfy goals, quality constraints, and softgoals. We can add hypothetical tasks to the r-net in Figure 4.6 so that no source nodes are goals, quality constraints, or softgoals. Figures 4.9 and 4.10 highlight two maximal consistent subnets of the resulting r-net. Each of these is a candidate solution, because (i) neither has goals, quality constraints, or softgoals as source nodes, and (ii) each includes the only mandatory requirement $\mathbf{g}(p)$.

Once we have found the candidate solutions, the question is how do they compare? We can establish that the two candidate solutions, denoted $\mathscr{S}_A$ (the subnet highlighted in Figure 4.9) and $\mathscr{S}_B$ (the subnet highlighted in Figure 4.10), have the following mandatory, optional, and preferred nodes:

- $\mathscr{S}_A$ (i) has $\mathbf{q}(p_4)$, is both an optional and a preferred requirement; (ii) has $\mathbf{s}(p_5)$ which is an optional requirement; and (iii) has $\mathbf{g}(p)$ which is a mandatory requirement.
- $\mathscr{S}_B$ (i) has $\mathbf{g}(q_3)$ which is a preferred node; (ii) has $\mathbf{s}(p_5)$, an optional requirement; and (iii) has $\mathbf{g}(p)$, a mandatory requirement.

The following comparison table gives the summary:

	prf : $\mathbf{g}(q_3)$	**prf** : $\mathbf{q}(p_4)$	**opt** : $\mathbf{q}(p_4)$	**opt** : $\mathbf{s}(p_5)$
$\mathscr{S}_A$	no	yes	yes	yes
$\mathscr{S}_B$	yes	no	no	yes

Each column in the comparison table is a criterion for the compari-

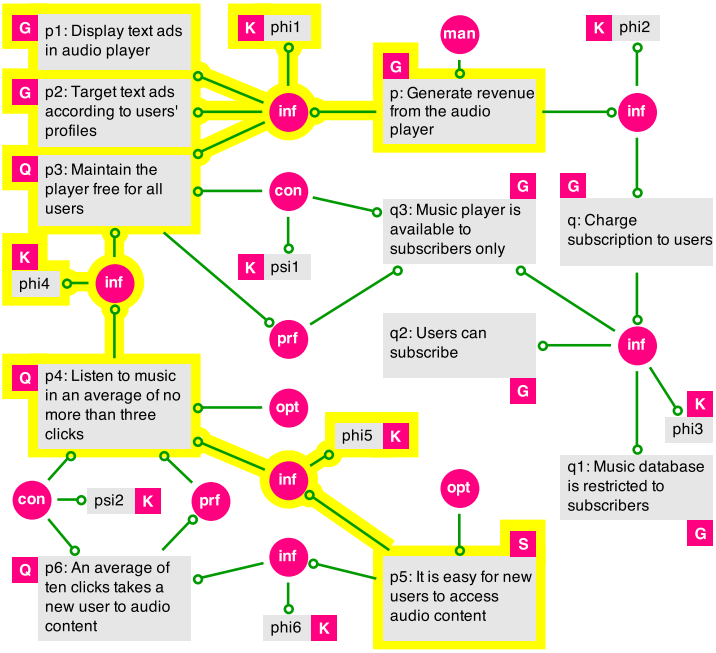


Figure 4.7: A consistent (sub)net is highlighted.

son of candidate solutions. Techne does not suggest how to make a total order over candidates in a comparison table.

4.5.5 Formalisation

Automating the search for candidate Solutions requires that the elements of Techne introduced so far obtain mathematically formal definitions. To sketch the formalisation, recall that a modelling language has four parts: (i) an alphabet of symbols, (ii) rules of grammar to combine symbols into sentences, (iii) a semantic domain with the objects of interest to the purpose of the language, and (iv) mappings from the symbols and sentences to the objects in the semantic domain. As I mentioned earlier, the first and second components are usually called *syntax*, the last two *semantics*.

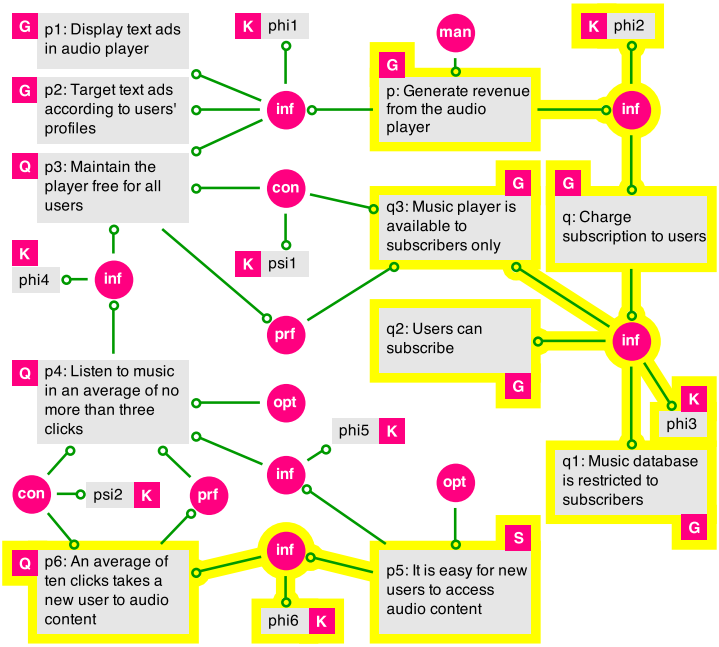


Figure 4.8: Another consistent (sub)net is highlighted.

R-net Alphabet

To draw r-nets, we used symbols for (i) atomic statements (indexed/primed p, q, r), (ii) complex statements (Greek letters) (iii) labels (**k**(), **g**(), **q**(), **s**(), **t**()), (iv) relations (**inf**, **con**, **prf**, **man**, **opt**), and (v) arrow-headed lines.

R-Net Grammar

Grammar is dictated by the CORE ontology for the use of labels, and the arity of relations for the use of relation symbols and lines. All allowed sentences are shown in Figure 4.11, and every r-net is exactly the finite set of elements shown in that figure.

In Figure 4.11, every p, q is an arbitrary atomic statement, every ϕ an arbitrary complex statement, and every **x** an arbitrary label. For **inf**, ϕ abbreviates "if $\mathbf{x}_1(p_1)$ and … and $\mathbf{x}_m(p_n)$, then $\mathbf{x}_{m+1}(q)$"; for **con**, ϕ is for "if $\mathbf{x}_1(p_1)$ and … and $\mathbf{x}_m(p_n)$, then contradiction". As

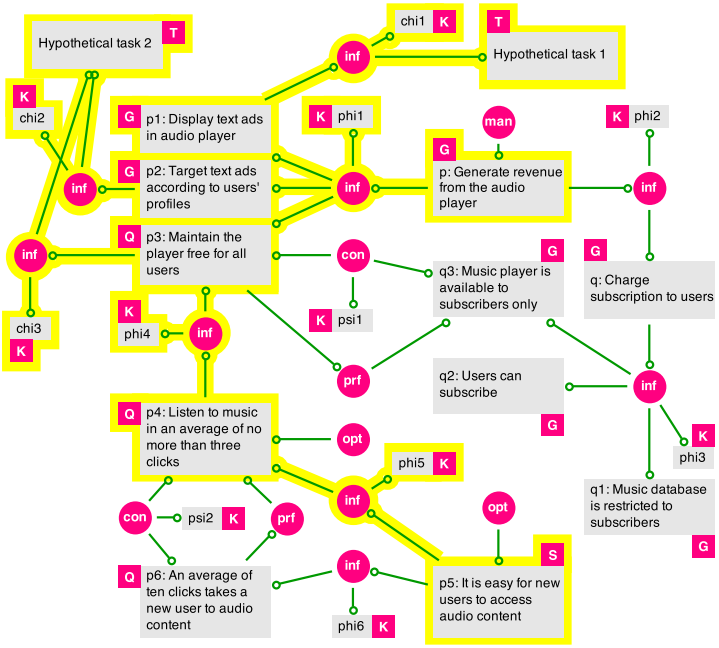


Figure 4.9: Candidate Solution r-net **A** is highlighted.

every complex statement refers to an assumption, it must have the label **k**().

Semantic Domain and Mapping

The elementary objects in the semantic domain of r-nets are propositions stating the properties of the system-to-be and its operational environment and the inference, conflict, preference, is-mandatory, and is-optional relations between them. Atomic and complex statements in the alphabet refer/map to these pieces of information, relation symbols map to relations, while sentences refer to combinations of the two. Following the statement of the requirements problem and as we want to avoid contradictory solutions, a candidate Solution is information which is (i) not contradictory and (ii) from which we can conclude that mandatory goals and quality constraints are satisfied.

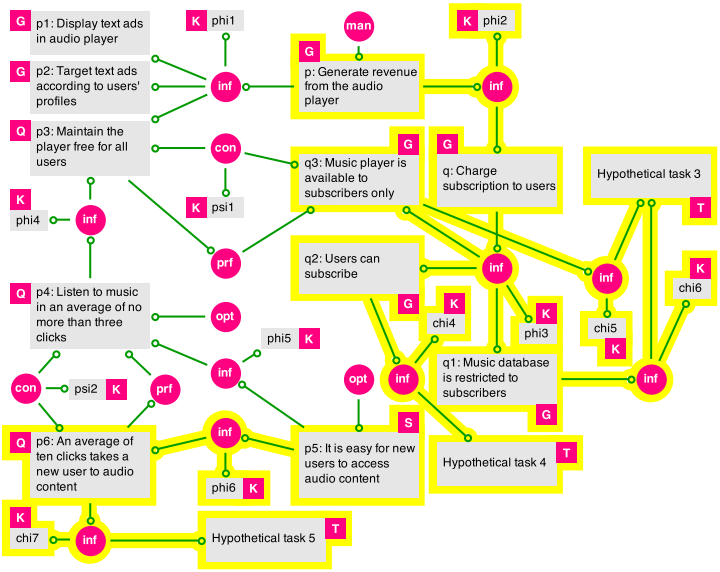


Figure 4.10: Candidate Solution r-net **B** is highlighted.

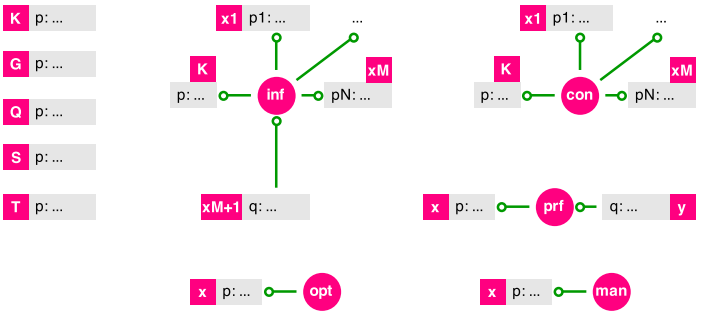


Figure 4.11: Allowed sentences in an r-net.

Proof-Theoretic Characterization of Candidate Solutions

To find candidate solutions, we need to find their counterparts in syntax, that is, those parts of r-nets which map exactly to candidate solutions in the semantic domain. As we will be comparing solutions after we find them, we leave out the information for the comparison

of candidate solutions (the preference, is-mandatory, and is-optional relations).

In syntax, this means that we focus not on a given r-net, but on its *attitude-free* variant: given an r-net $\mathscr{R}$, to make its attitude-free variant $\bar{\mathscr{R}}$, delete all **prf**, **man**, and **opt** nodes and all lines entering and leaving from these nodes from $\mathscr{R}$. $\bar{\mathscr{R}}$ contains only the atomic and complex statements, and the inference and conflict relations.

An $\bar{\mathscr{R}}$ can be seen as a set of proofs. To do so, we observe that our complex statements are sentences in the conditional if-then form in which the fragment after the *if* references requirements, while the one after *then* references either a single requirement, or contradiction (see Examples 4.5.3–4.5.7). We rewrite every complex statement ϕ in $\mathbf{k}(\phi)$ as a formula with conjunction and implication: every ϕ is such that either

$$\phi \equiv \bigwedge_{i=1}^{n} pl_i \rightarrow pl,$$

or

$$\phi \equiv \bigwedge_{i=1}^{n} pl_i \rightarrow \bot,$$

where every pl is some requirement (for example, $pl \equiv \mathbf{g}(q)$) and $\bot$ refers to logical inconsistency.

Figure 4.12 shows the sentences obtained by applying the said rules on the $\bar{\mathscr{R}}$ in Figure 4.3.

$$\frac{\mathbf{g}(\mathrm{p}_1) \quad \mathbf{g}(\mathrm{p}_2) \quad \mathbf{q}(\mathrm{p}_3) \quad \mathbf{g}(\mathrm{p}_1) \wedge \mathbf{g}(\mathrm{p}_2) \wedge \mathbf{q}(\mathrm{p}_3) \rightarrow \mathbf{g}(\mathrm{p})}{\mathbf{g}(\mathrm{p})}$$

$$\frac{\mathbf{g}(\mathrm{q}_1) \quad \mathbf{g}(\mathrm{q}_2) \quad \mathbf{g}(\mathrm{q}_3) \quad \mathbf{g}(\mathrm{q}_1) \wedge \mathbf{g}(\mathrm{q}_2) \wedge \mathbf{g}(\mathrm{q}_3) \rightarrow \mathbf{g}(\mathrm{q})}{\mathbf{g}(\mathrm{q})}$$

$$\frac{\mathbf{g}(\mathrm{q}) \quad \mathbf{g}(\mathrm{q}) \rightarrow \mathbf{g}(\mathrm{p})}{\mathbf{g}(\mathrm{p})} \qquad \frac{\mathbf{q}(\mathrm{p}_3) \quad \mathbf{g}(\mathrm{q}_3) \quad \mathbf{q}(\mathrm{p}_3) \wedge \mathbf{g}(\mathrm{q}_3) \rightarrow \bot}{\bot}$$

Figure 4.12: The $\bar{\mathscr{R}}$ from Figure 4.3 rewritten as four proofs.

Attitude-free r-nets are sets of proofs of the formal system in which the atoms pl of the alphabet are symbols for requirements (for example, $\mathbf{g}(p)$), the only allowed sentences are $\bigwedge_{i=1}^{n} pl_i \rightarrow pl$ and

$\bigwedge_{i=1}^{n} pl_i \rightarrow \bot$, and the only rule of inference is modus ponens. Given a set of such requirements and sentences denoted $\bar{S}$ and $x \in \{pl, \bot\}$:

1. $\bar{S} \mid\sim_{\tau} pl$ if $pl \in \bar{S}$, or
2. $\bar{S} \mid\sim_{\tau} x$ if $\forall 1 \leq i \leq n$, $\bar{S} \mid\sim_{\tau} pl_i$ and $\mathbf{k}(\bigwedge_{i=1}^{n} pl_i \rightarrow x) \in \bar{S}$.

The consequence relation $\mid\sim_{\tau}$ is sound w.r.t. standard entailment in propositional logic, but is incomplete in two ways: it only considers deducing positive atoms, and no ordinary proofs based on arguing by contradiction go through, thus being paraconsistent.

The consequence relation leads us to the following conception of the *candidate solution* concept: Given an $\mathscr{R}$ with all of its domain assumptions in the set **K**, tasks in **T**, goals in **G**, quality constraints in **Q**, and softgoals in **S**, *a set of tasks* $\boldsymbol{T}^*$ *and a set of domain assumptions* $\boldsymbol{K}^*$ *are a* ***candidate solution*** *to the requirements problem of* $\mathscr{R}$ *if and only if (i)* $\boldsymbol{K}^*$ *and* $\boldsymbol{T}^*$ *are not inconsistent, (ii)* $\boldsymbol{K}^*, \boldsymbol{T}^* \mid\sim_{\tau} \boldsymbol{G}^*, \boldsymbol{Q}^*$, *where* $\boldsymbol{G}^* \subseteq \boldsymbol{G}$ *and* $\boldsymbol{Q}^* \subseteq \boldsymbol{Q}$, *(iii)* $\boldsymbol{G}^*$ *and* $\boldsymbol{Q}^*$ *include, respectively, all mandatory goals and quality constraints, and (iv) all mandatory softgoals are approximated by the consequences of* $\boldsymbol{K}^* \cup \boldsymbol{T}^*$, *so that* $\boldsymbol{K}^*, \boldsymbol{T}^* \mid\sim_{\tau} \boldsymbol{S}^{\boldsymbol{man}}$, *where* $\boldsymbol{S}^{\boldsymbol{man}}$ *is the set of mandatory softgoals.*

The candidate Solution concept leads us in turn to a more precise formulation of the requirements problem: ***Given*** *an r-net* $\mathscr{R}$, ***find*** *its candidate solutions.* Once candidates are found, the comparison table can be constructed in the straightforward way so that they can be compared. It is beyond the scope of this paper to give guidelines on how to rank candidates on the basis of the comparison table. Figure 4.13 highlights the members of these two sets and thus a candidate Solution for the r-net from Example 4.5.8. Note that Figures 4.9 and 4.10 highlight candidate Solutions and all of their consequences.

A note on expressiveness: observe that if we treat the *pl*s as atomic propositions, then an r-net is a Horn theory (every formula has at most one positive atom), which is known to be less expressive than full propositional logic, let alone predicate logic. Among others, there is no provision for world knowledge that is disjunctive (for example, composite *pl* like $p \vee q$), but we can express exclusive disjunction (for example, in Figure 4.3, $\mathbf{g}(p)$ is refined by either $\mathbf{g}(q)$ or by the conjunction of $\mathbf{g}(p_1)$, $\mathbf{g}(p_2)$, and $\mathbf{g}(p_3)$). There is also no provision for inference nodes that might use lemmas as $\mathbf{k}(\phi)$, which might lead to case-based reasoning. On the other hand, if we consider only attitude-free r-nets, the problem of finding candidate Solutions can

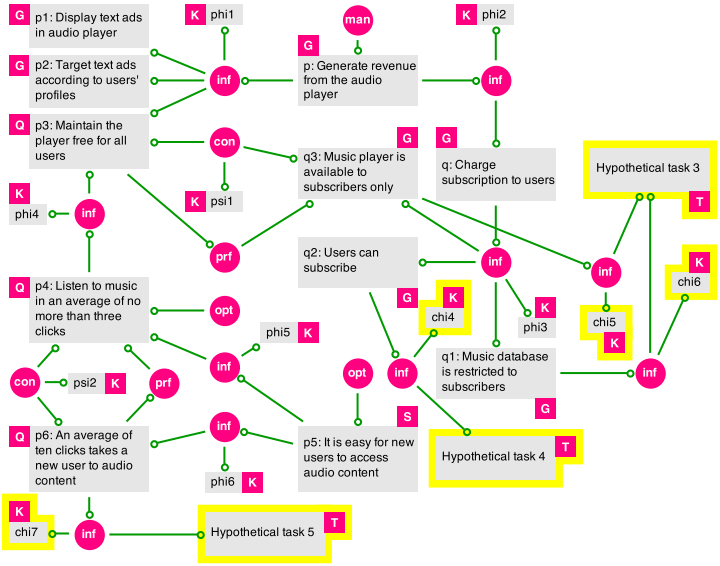


Figure 4.13: Members of $\mathbf{T}^*$ and $\mathbf{K}^*$ are highlighted.

be reduced to variants of solving non-standard reasoning problems in logic, such as abduction ("What tasks are needed to ensure the mandatory goals?"). Interestingly, it is known that Horn abduction is one level lower in the polynomial complexity hierarchy than abduction with full propositional logic, so our version of Techne has lower expected computational cost — a typical expressiveness/complexity trade-off. Only extensive practical experience in modelling will show whether more expressive power is needed.

4.5.6 Comments

My involvement in making Techne inevitably colours what I think of it. My own motives for making Techne were to make the language which has the fewest possible concepts and relations, and still achieves two aims. Firstly, that it has concepts which fit the CORE ontology and can produce models which represent CORE Problems and CORE Solutions. Secondly, that it can represent common relations in Requirements Modelling Languages which preceded it, such as goal refinement and conflict from KAOS, task decomposition from

i-star, and so on. It was also interesting to make a language which does not come with a predefined visual syntax.

In contrast to i-star, which introduced the modelling of dependencies in coordinated work, Techne introduced no new relationships. Concepts in CORE, and thus in Techne, as well as the relationships in Techne can all be seen as a synthesis of various ideas discussed previously in Requirements Engineering research. For example, quality constraints and softgoals are very much related, and draw on research on non-functional requirements [108, 29]. KAOS introduced the concept of "goal" to Requirements Engineering, and it remains unchanged in Techne. There is novelty in the treatment of conflicts, and in having a paraconsistent syntactic consequence relation in Techne, which lets it tolerate logical inconsistency when drawing conclusions from models.

Techne 2 [84] was made after trying to apply Techne to the problem of representing Problems and Solutions in cases when requirements change over time. The challenge is that change of requirements means change of the Problem to solve, and thus necessarily the design of, or search for a new Solution. This leads to the need for additional tools in a language, such as the ability to represent uncertainty in the degree to which a requirement, say a goal, will be satisfied during some period of time. It also requires being able to show what will happen, in case a goal fails to be satisfied, or some conditions in the environment fail to hold. Overall, it leads to a language which can be used to say more than a Techne model can, but at the cost of having to learn a more complicated language and use more complicated models.

Chapter 5

Requirements Problem Solving Cases

This Chapter presents several genuine Problem Situations. For some, it outlines the outcomes of only the very first steps of Requirements Problem Solving. One case is used to illustrate the languages defined in this book, and the others are given as exercises. In later Chapters, there is typically a definition of a Requirements Modelling Language, and models in that language will represent information from the cases in this Chapter. Confidentiality was important in most cases. The names of stakeholders and companies are removed and replaced by generic ones. Name of each case is made of the name of a city which was somehow related to the case, and of the domain of expertise of the case. Cases are called Brussels Law, Copenhagen Sports, Dubai Telecom, London Lights, and London Ambulance. London Ambulance is a classical case from Requirements Engineering research, and it is often used to illustrate Requirements Modelling Languages. The other cases are inspired by my experience in Requirements Problem Solving.

5.1 Brussels Law

This problem occured at a notary business in Brussels. The initial information about it came from a legal professional, shortly after she passed all the necessary examinations to become a civil law notary in Belgium. Following the country's laws and conventions, and in order to become a practicing notary, she invested in purchasing the license and the business of a retiring notary in Brussels.

For all practical purposes, you can see the notary business as a small company, even though it is subject to special regulation (such as limitations on marketing), but which are not relevant for this discussion. At the time, the business had 10 employees. These employees assist the notary in performing her principal task, summarily described as follows by the Belgian federation of royal notaries (Fédération Royale du Notariat Belge):

> "A civil law notary's task is to advise persons who wish to conclude agreements in areas as diverse as real estate, family or business matters. A civil law notary informs his or her clients about their rights and duties, and about the consequences – legal, financial and tax-related – of their commitment. The civil law notary listens to his clients' needs and advises them on all these areas."

The transition from the retiring notary to the new owner was a challenge. The notary observed that employees were making many decisions on their own when interacting with clients. While some degree of autonomy is encouraged, the aim for the notary is to give the best possible advice, and this requires notary's own expertise and involvement.

This overall issue motivated a round of interviews with a sample of the employees, to see if and how they perceive this issue, if they see any related ones, as all this would help understand which problem to solve first, and how best to approach its resolution.

The interviews were organised around topics, which included explaining the work the individual does at the office, and her impression of problems that there may exist in her own work, and in work with colleagues. Six employees were interviewed. Each interview lasted approximately 30 minutes.

The information collected at the interviews was used to produce a summary of the interviews and a terminology, which defines the

terms which have a specific meaning within the notary profession, or within the office itself.

5.1.1 Terminology

The following terms have a specific definition within the Brussels Law problem. They were identified after, and from the information elicited in the interviews.

- *Notary Business:* Notary office lead by the Notary.
- *Product:* Legal document produced and signed by the Notary for a Client, in exchange for a fee.
- *Pre-Product:* Document prepared by a Notary Assistant or Notary, and which has not yet been signed by the Notary.
- *Client:* Legal entity purchasing Products from the Notary Business.
- *Notary:* The individual holding the license to perform notary duties, who signs legal acts, is legally responsible for the content of the legal acts, owns and leads the Notary Business, communicates with Clients, prepares legal acts.
- *Prior Notary:* Notary who preceded the current Notary. Communicates with Clients, to ease Clients' transition to the new Notary.
- *Notary Assistant:* Prepares Products for review, validation, and signing by the Notary.
- *Secretary:* Manages the expedition of Products, handles administration, incoming phone calls, incoming and outgoing mail, finds archived legal acts, etc.
- *Accounting Assistant:* Maintains accounting information through accounting software.

5.1.2 Interviews Summary

Interviews were conducted with the Notary Business employees, including three Notary Assistants, two Secretaries, and one Accounting Assistant.

The summary below is organised by topic that arose during one or more interviews. Topics are neither ordered by priority, nor another criterion.

1. Advising Clients:
   (a) All three Notary Assistants highlighted that they appreciate giving advice to Clients.
   (b) By advice, Notary Assistants seem to mean information related to, but not necessarily included in the Product made for the Client.
   (c) When asked to point to sources of advice, Notary Assistants indicate their own individual experience.
   (d) When asked if they validate the advice with the Notary, Notary Assistants avoid the question, leaving the impression that there is little to no validation of the advice.
2. Methods for delivering advice:
   (a) Notary Assistants deliver advice to Clients in the following ways, from the most frequent to the least frequent: email, phone, live meetings, mail.
   (b) All Notary Assistants indicated that email is the most important form of communication, in order to keep a trace of the exchange of information with the Client.
3. Supervision of Notary Assistants in Client interaction: When asked when the Notary intervenes in the Notary Assistant's work with a Client, the Notary Assistants indicated that the Notary intervenes predominantly at the end of the work with the Client, for the transformation of Pre-Product into Product.
4. Client redirection to Notary Assistants: Client who calls Notary Business by phone, and asks to speak to a particular Notary Assistant, is redirected by a Secretary to that Notary Assistant.
5. Affirmation of position:
   (a) In all six interviews, every interviewee asserted their relevance and importance for Notary Business. Every interviewee asserted their importance on their own initiative, as no question was asked which would suggest future changes in positions and responsibilities.

(b) One Notary Assistant and one Secretary suggested that they are the only ones able to discharge the responsibilities they currently hold.

6. Understanding of own position and responsibilities:

   (a) One Notary Assistant stated that, for her Clients, she is the Notary.

   (b) All Notary Assistants see Clients as their own, not as those of the Notary Business.

   (c) The newly hired Secretary suggested by herself that - compared to her prior work as secretary of top management - the positions, responsibilities, and hierarchy are not clear at Notary Business.

7. Transition from Past Notary. Notary Assistant, the spouse of the Past Notary, highlighted the following:

   (a) There is more work compared to period when Prior Notary lead the office.

   (b) Coordination of work is worse now than when Prior Notary lead the office.

   (c) Flow of information between employees at Notary Business is less efficient now than the fully paper-based system was when Prior Notary lead the office.

   (d) This Notary Assistant had the impression that Notary is trying to do too much at the same time, and specifically that Notary is trying to do both a Notary's work and the work of one Notary Assistant (i.e., the making of Pre-Products).

8. Expedition of Products:

   (a) There are rules defined in law, for whom to distribute a Product (i.e., whom to send a legal act once it is signed by the Notary).

   (b) One Secretary suggested that she is the only person at the moment who knows and applies these rules.

   (c) When this Secretary was absent, no one could perform her work.

9. Documentation of knowledge: When asked where she is learning the processes and rules of work from, the newly hired Secretary suggested that there is no documentation and that she makes her own notes on rules and processes.

5.1.3 Problems

In a summary, interview topics suggest the presence of the following problems:

- Unclear position responsibilities and hierarchy relations;
- Absence of processes and rules to govern the interaction with Clients, the making of Products, and the expedition of Products;
- Absence of processes and rules for documenting knowledge and training of employees;
- No uniqueness and innovation in Products and interaction with Clients.

and these problems can have the following interrelated consequences:

- Damage to the Notary Business brand;
- Inconsistent quality of Products;
- Unusual dependence on individual employees, and thereby risk of inefficiency in their absence;
- Difficulties in management, in absence of means to preserve and transfer knowledge between employees.

Below are more details on which problem is related to which consequences:

1. No explicit definition of authority relations and position responsibilities. Possible consequences include:
    - Employees do not update the Notary of their tasks and deadlines;
    - Employees position themselves as the main point of contact for Clients at Notary Business;

- Employees give Clients advice which has not been validated by the Notary.

2. No processes and rules to govern the interaction between Clients and Notary Business. Possible consequences include:
   - Notary Assistants interact autonomously with Clients, so quality of interaction cannot be monitored, influenced, and standardised;
   - Notary is not informed of communication between Client and Notary Assistant and cannot influence and evaluate the quality of that interaction, when that communication happens via:
     - Email, as Notary is not included in CC of emails sent by Notary Assistants to Clients;
     - Phone, as Secretaries transfer Clients directly to a Notary Assistant;
     - Meeting outside the Notary Business office.
3. No processes and rules for monitoring and reporting on delivery of Products to Clients. Possible consequences include:
   - Task lists of Notary Assistants are obscure to the Notary;
   - Notary Assistants fully manage their task lists, and do not report on them.
4. No processes and rules for the preservation of knowledge and no transfer of knowledge between employees. Possible consequences include:
   - Knowledge stays with individuals, who thereby make Notary Business dependent on them;
   - It seems that rules or procedures in work are not documented, and have not been documented while Prior Notary was leading the office;
   - There are no rules to govern training between employees. It is up to the employee being trained, to document or otherwise formalize and learn the knowledge obtained from another employee.

5. No processes and rules for communicating new and changed rules and processes to employees. Possible consequences include having no clear communication to employees on what rules and processes should be followed.

6. No uniqueness in interaction with Clients and no Product innovation. Possible consequences include:

    - Notary Assistants act autonomously in interaction with Clients;
    - Notary is not visible to Clients.

5.2 Copenhagen Sports

This case involves a small sports company in Copenhagen. The company makes and sells online coaching software which enables sports coaches to train recreational, amateur, and professional athletes at a distance.

For the coach, the software is a tool to create training plans and training instructions, assign them to individual athletes, and receive their feedback. For the athlete, the software is the place to find training instructions and her training plan, to record training progress, as well as to keep a diary of sports and related activities which can be relevant for the coach, when she designs the training for the athlete (such as which foods the athlete ate, if they travelled on some particular day, etc.). For both coaches and athletes, the software provides data visualisation tools, as well as other means designed to facilitate their communication and interaction.

The majority owner, himself a coach, was interested in how the software could help coaches with two activities that take considerable time, namely, invoicing and payments. Up to that point, all invoicing and payment between a coach and an athlete was the responsibility of the coach. In case the coach worked for, say, a gym or a sports club, then that legal entity was invoicing and receiving payments from the athletes. Someone at the gym or sports club still had to manage invoicing and payments, and the coaching software offered no help.

This initial idea led to an interview with the owner, in his role as a coach who uses the software with his athletes. The rest of this section gives the terminology required to understand the interview summary, and then the problem description.

5.2.1 Terminology

The following terms have a specific definition within the Copenhagen Sports case. They were identified after, and from the information elicited in the interview with the coach.

- *Coaching Company:* Company which makes and sells the online coaching software.
- *Customer Invoice:* Invoice to an individual who should pay to Coaching Company.
- *Custom Customer Invoice:* Customer Invoice which should be made using a non-standard template, defined by the customer, and, or include information which other Customer Invoices do not include.
- *Company Invoice:* Invoice to a company who should pay to Coaching Company. It includes specific company information (for example, company registration number) which is not included in Customer Invoice.

5.2.2 Interview Summary

The aim of the interview was to understand what invoicing and payments tasks a coach usually has to do. Without this information, any proposed solution may entirely ignore constraints which these tasks are currently subject to.

The summary of the interview is given below. It lists first the invoicing tasks, and then the payments tasks that a coach typically does.

Invoicing tasks include the following:

1. *Subscription invoicing* focuses on having customers pay for their monthly subscriptions to Coaching Company services. Subscription Invoicing involves the following tasks:
   (a) Update spreadsheet with names of all customers who should receive Customer Invoice for current month; a coach does this;
   (b) Make all Customer Invoices for the current calendar month. Make Custom Customer Invoices for all customers who requested them; coach's accountant does this;

(c) Send all Customer Invoices and Custom Customer Invoices to all customers at end of current calendar month; the coach's accountant does this;

(d) Check, every Monday, if every sent Customer Invoice has been paid, by checking Coaching Company bank account and matching incoming payments with Customer Invoice numbers; the coach does this;

(e) For each Customer Invoice which has not been paid, inform relevant OOB Coach to remind the customer to pay; the coach does this;

(f) Send reminders to Customers who have not paid, that they have a Customer Invoice to pay.

2. *Event invoicing* focuses on having companies pay for Coaching Company services, which are not paid by subscription. Event invoicing typically involves these tasks:

   (a) Elicit instructions from client for Company Invoice;

   (b) Make Company Invoice;

   (c) Send Company Invoice;

   (d) Check if Company Invoice is paid;

   (e) If Company Invoice is not paid, then remind client to pay Company Invoice.

   The coach does all the tasks above.

Accounting tasks include the following:

1. Keep a record of all Invoices sent.

2. Keep a record of all invoices paid.

3. Record every expense of Coaching Company, and store proofs of expenses.

4. Keep a record of all invoices received and to be paid by Coaching Company.

5. Pay all invoices received and to be paid by Coaching Company.

6. Keep a record of all payments of invoices received and paid by Coaching Company.

7. Provide records of all Coaching Company expenses and proofs of expenses to danish accountants of Coaching Company.

The coach does all the accounting tasks above.

5.2.3 Problem

The owner of the Coaching Company asked that the coaching software should be changed to support coaches with the invoicing and payment tasks described in the interview. In one or several next releases, the software should do at least some of the tasks in place of a coach, or in some other way facilitate these tasks to a coach.

5.3 Dubai Telecom

This case involves a company which specialises in making business-to-consumer software products for Telecommunications Service Providers (TSPs), such as companies offering prepaid and contract mobile telephony and data services. The products are used by TSPs to run their consumer websites, where they promote products and services, to run their online services, such as online technical support, to sell their services and products, and so on.

The company made a new product, called TSPAppDev in this book. The product is software which allows TSP's customers to make apps for mobile devices through an intuitive interface, to minimise the time to make an app and appeal to customers who need to make simple apps quickly. The overall idea is that the TSP buys the software, lets its customers use it through a free trial period, and then charges a monthly subscription for customers who wish to continue to use it.

The Chief Executive Officer of the company wanted to have a clear plan of action, for how to deliver TSPAppDev to a TSP which buys it. This motivated interviews with him, and subsequently with the product director. The rest of the section gives the information elicited after these interviews.

5.3.1 Interviews Summary

CEO Interview

The interview with the CEO resulted in the information about the main lifecycle phases of the TSPAppDev product. This information is

below.

TSPAppDev is directed at telecommunications companies, TSPs, interested in introducing new revenue streams and strengthening loyalty of corporate and residential customers.

TSPAppDev is delivered in three phases:

- Phase 1 - TSPAppDev Competition: TSPAppDev is delivered to enable a 3 months or longer Competition, the goals of which are to:
  - Increase brand awareness in local market;
  - Promote local Mobile App development Community;
  - Collect market information on Business and Residential Users interest in Mobile App Development.
- Phase 2 - Campaign Evaluation: Analysis of data acquired during Competition.
  - If significant market potential is recognised in the Competition, the project is transferred into Service Setup;
  - Alternatively 3-month Competition is repeated or Client exits the process;
  - Data collected and analysed includes: number of registered users, number of created and submitted Mobile Apps; number of Mobile Apps ready for distribution in mobile app stores, number of downloaded Mobile Apps, number of social network fans, number of Competition website visitors.
- Phase 3 - TSPAppDev Service: TSPAppDev is delivered under a software as a service model to TSP, in order to enable TSP's residential and business customers to make Mobile Apps.

In addition to making free Mobile Apps, residential and business customers can create premium Mobile Apps. Premium Mobile Apps create new revenue streams for TSP, through per-Mobile App setup and monthly subscription fees paid by residential and business customers.

Product Director Interview

The product director highlighted that it is important to understand what the product does for the TSP and its customers. This should suggest what needs to be done, in order to deliver it to the TSP, and set it up so that it can be used by TSP's customers.

TSPAppDev Competition works as follows:

- Competition is a competition in making Mobile Apps using TSPAppDev;
- Any user in TSP country can participate;
- To participate, users register to Competition website, create, and submit Apps;
- Users can create free Apps only. Free Apps are limited in three ways:
  - Include advertising managed by TSP;
  - Publisher is TSP, not the author of the App;
  - Apps are created using limited set of TSPAppDev features;
- TSP reviews submitted Apps, publishes selected Apps to TSP?s and global Apps stores;
- Jury and users rate submitted Apps, and select winners;
- Winners receive awards at an award ceremony.

TSPAppDev Service works as follows:

- TSP uses TSPAppDev Service to enable its Residential and Business Customers to make and distribute Free and Premium Mobile Apps;
- For the countries where TSP purchased TSPAppDev Service, no other telecommunications company can purchase TSPAppDev Service;
- TSPAppDev is distributed under a software as a service model;
- All Customers can make Free Mobile Apps;

- Every residential customer pays for each Premium Mobile App a monthly subscription fee. When purchasing each premium App, the residential customer pays the equivalent of 6 months' subscription, which pays for her first 6 months. After the first 6 months, the residential customer pays the subscription fee per month;

- Every business customer of the TSP pays for each premium Mobile App a monthly subscription fee. When purchasing each premium App, the residential customer pays the equivalent of 6 months' subscription, which pays for her first 6 months. After the first 6 months, the residential customer pays the subscription fee per month.

5.3.2 Problem

Given the information from the interviews, the problem is to define how TSPAppDev should be delivered to a TSP which purchased it. Who should do it at the company? How? Are TSP's employees involved? If yes, which expertise do they need to have? When are they involved? What for? All such questions had to be answered.

5.4 London Lights

A small company designs products in the UK, manufactures them overseas, and sells them internationally via its own website and via physical distributors. The company's Managing Director was interested in having software, custom or otherwise, which would help keep track of the various steps and tasks in the development of each new product.

This lead to interviews with the Managing Director and the Creative Director, to understand the new product development process, and then think about the requirements for the software.

5.4.1 Terminology

- *New Product:* Product that London Lights plans to, but has not yet started selling.

- *First Sample:* A first and preliminary model of the New Product.

- *Final Sample:* A final model of the New Product, accepted by London Lights.
- *Provisional Launch Date:* Preliminary date when the New Product will become available for sale to consumers.
- *Definite Launch Date:* Date when the New Product will become available for sale to consumers.
- *Product Manufacturer:* Individual representing the company capable of manufacturing the New Product for London Lights.
- *New Product Brief:* Document specifying the requirements that the New Product should satisfy.
- *Manufacturer Estimate:* Document by which the Product Manufacturer responds to New Product Specifications.
- *New Product Pricing:* Document defining the prices for the New Product, for all Sales Channels.

5.4.2 Interviews Summary

The interviews led to the following description of the new product development process.

1. Creative Director and Managing Director discuss New Product ideas;
2. Managing Director and Product Developer perform research on the New Product;
3. Creative Director and Managing Director:
    - Create New Product concepts;
    - Request Product Developer to evaluate New Product concept feasibility;
4. Product Developer:
    - Evaluates New Product concept feasibility;
    - Delivers feasibility evaluation to Managing Director and Creative Director;

5. Creative Director, Managing Director, and Product Developer narrow down the requirements to include in the New Product Brief;
6. Managing Director:
    - Produces the New Product Brief;
    - Obtains from the Creative Director the approval of the New Product Brief;
    - Sends the New Product Brief to Product Developer;
7. Managing Director and Creative Director name a Product Designer;
8. Product Designer:
    - Produces New Product design concepts;
    - Presents design concepts to Creative Director and Managing Director;
    - Adapts design concepts until Creative Director and Managing Director approve a concept;
9. Product Designer:
    - Produces prototype of New Product design concepts;
    - Presents design concept prototype to Creative Director and Managing Director;
    - Adapts design concept prototype until Creative Director and Managing Director approve the prototype;
10. Managing Director updates and sends New Product Brief to Product Manufacturer;
11. Product Manufacturer responds to Managing Director on New Product Brief;
12. Creative Director, Managing Director, and Product Developer revise, if needed the New Product Brief;
13. Managing Director sends revised New Product Brief to Product Manufacturer;

    Note: Steps 10 to 13 are repeated until the Product Manufacturer can provide the Manufacturer Estimate to the Managing Director;

14. Product Manufacturer delivers Manufacturer Estimate to the Managing Director.

    Manufacturer Estimate includes estimates of:

    - Product development cost;
    - Product development timeline;
    - Minimal order size;
    - Estimated unit cost.

15. Creative Director and Managing Director decide if to accept the Manufacturer Estimate;

    - If no, contact another Product Manufacturer and go back to Step 6;
    - If yes, go to next Step;

16. Managing Director:

    - Receives component samples from Product Manufacturer;
    - With Creative Director, reviews component samples;
    - Asks Product Manufacturer to produce First Sample;

17. Product Manufacturer:

    - Produces First Sample;
    - Delivers First Sample to Managing Director;

18. Creative Director, Managing Director, and Product Developer give feedback on the sample to the Product Manufacturer;

19. Product Manufacturer delivers the revised Manufacturer Estimate to the Managing Director;

20. Managing Director requests a new sample from the Product Manufacturer;

21. Product Manufacturer delivers a new sample to Managing Director.

    Note: Steps 19 to 21 are repeated until Managing Director approves a sample and that sample becomes the Final Sample;

22. Managing Director approves the Final Sample by sending email to Product Manufacturer and Creative Director;
23. Product Manufacturer delivers final Manufacturer Estimate to Managing Director;
24. Managing Director:
    - Receives 30 units of the Final Sample form Product Manufacturer;
    - Sends units of Final Sample for photography;
    - Sends units of Final Sample to key Clients;
25. Managing Director submits the Final Sample to certification:
    - For CE (Conformité Européenne) mark, required for sales to European Economic Area;
    - For UL (Underwriters Laboratories) mark, required for sale to USA;
    - For JIS (Japanese Industrial Standards) mark, required for sale to Japan.
26. Managing Director presents packaging to Product Manufacturer. Cases:
    - Case 1: Product Manufacturer approves packaging. Go to next step;
    - Case 2: Product Manufacturer requests changes to packaging.
      - Managing Director organizes that packaging be changed;
      - Go to Step 26;
27. Finance and Operations Director:
    - Sets the Provisional Launch Date;
    - Communicates Provisional Launch Date to Managing Director;
28. Managing Director:
    - Approves the Provisional Launch Date;

- Sends the Provisional Launch Date to:
- Board;
- Sales Manager;
- Product Developer;
- Creative Director;
- Finance and Operations Director;
- Accounting and Finance;
- Defines New Product Pricing;
- Obtains approval of New Product Pricing from the Board;
- Sends final New Product Pricing to Board, Finance and Accounting.

5.4.3 Problem

Given the current new product development process, how could software help track various steps and tasks in that process? How would that software influence the process itself? Would the process have to change, and in what ways, to accommodate the software?

5.5 London Ambulance

This case draws on the London Ambulance Service's Computer-Aided Dispatch (LASCAD) system [3], which has often been used in Requirements Engineering to illustrate Requirements Modelling Languages [76, 145, 146, 99]. The information in this case borrows borrows Beynon-Davies' presentation of LASCAD [12].

LASCAD was intended to replace manual dispatching of ambulances to incident locations. A manual dispatching system consists of the following [12]:

> *"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 [12]:

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 problem in the case is to design the computer-aided dispatch system in an environment where dispatching is done manually.

Although there is relatively little information above, it 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.

Chapter 6

Checklists, Templates, and Services for Requirements Modelling Language Design

This Chapter presents three simple tools, called Language Checklists, Templates, and Services, and explains why and how they are used in this book. They helped make it manageable to define many languages, compare them, and carry their features from one to another. Language Checklists give items to include in a definition of an Requirements Modelling Language and its parts. Each Checklist suggests a Template, so that when it is relevant to define a Requirements Modelling Language or its part according to a Checklist, then there is a corresponding Template to fill out. Services describe specific problem solving tasks that a language and relevant algorithms automate for a human problem solver. I use Services to describe the purpose a language and its parts in problem solving, Checklists to avoid missing important parts of definitions, and Templates to standardise the presentation of definitions.

6.1 Problem Solving Services

Let *Q* denote a question, such as, for example, "Which requirements are satisfied in the given Model?". To answer this question, you need to have elicited requirements, found ways to satisfy them, represented all this in a model, and then analyse the model to see if it answers the question.

But the question itself has nothing to do with how you made the model, that is, the elicitation and any treatment of information that was needed to make the model. The question is not influenced by how the model was made. Moreover, the question is not specific to one particular Requirements Modelling Language. The same question can be asked for various models, made in various languages, as long as these languages have a notion of "requirement" and "satisfaction". The question would otherwise be meaningless, but again, only for those languages which fail to distinguish requirements in a model from other information in the model, and to distinguish when something in the model can be called "satisfied" and when it cannot.

Answering the question *Q* may be an ill-structured problem, even when you do have a model. This is the case precisely if, for example, the language of the model does *not* give the exact conditions that have to hold, in order to say that, without a doubt some requirement in that model, and according to that model, is satisfied. In other words, "being satisfied" is not precisely defined in the language.

In some languages, however, *answering question Q will be a well-structured problem.* In other words, you will know the steps to take and the tools to use in order to answer *Q* when given *any* model in that language. And *if someone else applies the same steps and tools according to instructions, then they will reach the same answer for the same model.*

In some cases, these languages may even be such that you or anyone else can apply algorithms to models, and automatically obtain the answer.

The interesting conclusion of the above is that you can describe a language and algorithms for Requirements Problem Solving by the questions they can answer for a human problem solver.

In the rest of the book, I will say that a language, and if needed algorithms, that is, Artificial Intelligence for Requirements Problem Solving, have Services, and each Service is a question that the AI can answer.

Problem Solving Service

Definition 6.1.1. *Problem Solving Service:* A Requirements Modelling Language has the Problem Solving Service *X*, or simply the Service *X*, if answering the question *X* for any reasonably-sized model in that language is a well structured problem.

The definition is loose. Services cannot be used to fully describe *all* that a language can do for a human problem solver. Some people will be more inventive than others, and perhaps manage to use a language to answer more questions than others would think of. For example, when does a model have a reasonable size? I do not have a definite answer to this, and if I did, I would have proposed a better definition.

But despite its obvious limitations, Services are interesting when used *to guide the design of a Requirements Modelling Language*. This is what I use them for in this book. It is an important role. Services can be used to justify features a language has, the concepts and relations in its ontology, its formalisation, etc.

The introduction and use of Services is motivated by the assumption mentioned in Section 4.2, that an Requirements Modelling Language should influence how one thinks about and solves Problems. More specifically, I will assume that an Requirements Modelling Language will effectively do so, if it can *do* something for its user, the human problem solver, that is, *if its user can delegate part of the problem-solving effort to the Requirements Modelling Language*.

You can think of it this way: there is a language user, a person who needs to solve an Problem, and suppose that there is software, which she uses to make Requirements Models. To find the solution to the Problem, as well as to properly formulate the Problem 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.[1] 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 Requirements Modelling Language*, and the language has to say what its models are, and how to process them to answer questions.

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

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 Service, even if they have very different components and work in different ways to answer the corresponding question.

How I define and use Services will become clearer in and after Chapter 7, when I start defining the first Requirements Modelling Languages specific to this book.

6.2 Checklists and Templates

A Checklist will list questions that you need to answer, in order to define, for example, a relation or a category that is used in a formal

[1] 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.

language. For example, to define a relation, it is necessary to say what its domain is, what rules every instance of that relation has to satisfy (such as those due to the relation being transitive, for example), its arity, and so on.

Template

There will be various Checklists in this book, and each is related to recurring components of a Requirements Modelling Language. For example, Requirements Modelling Languages typically provide categories for different kinds of elicited information, and there is a Checklist that suggests what a definition of a category has to say. Checklists are not exhaustive, but are instead used to ensure that basics are covered.

When there is a Checklist that recommends what goes in a definition, the natural next step is to define a corresponding Template to use when writing the definition. The Template includes slots, which if adequately filled out make sure that the definition answers the questions in the Checklist.

Checklist

There is a Template for every Checklist in this book. There are two primary uses of the Templates, one being to standardise the presentation of languages and their modules in the book, and the other to make it easier to present the information that the corresponding Checklist asks for.

6.3 Language and Module Names

Every language defined in this book 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 7.3 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.

Language Module

For example, a module name for one of the language in Section 7.4 is L.(r.inf.pos, r.inf.neg, f.map.abrel.g). This says that the language is made of three modules, denoted by r.inf.pos, r.inf.neg, and f.brel2g.

Each language module in the book has a unique abbreviation, and those abbreviations are used to form the module names of languages. The point is to know what modules a language includes, simply by looking at its module 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 [77].

Chapter 7

Relations

This Chapter 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 Requirements Engineering concerns, called influence and rationale below, which have usually been addressed via specialised relations in Requirements Modelling Languages. More specifically, the Chapter 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 7.2)*

2. *How to define oft-needed relations in such a way, that they can be reused when defining new Requirements Modelling Languages? (Section 7.3)*

3. *How to represent that satisfying some requirements influences the satisfaction of others? (Section 7.4)*

4. *How to represent the rationale for design decisions? (Section 7.5)*

5. *If a Requirements Modelling Language includes several relations, then how to avoid errors in using these relations together? (Section 7.6)*

7.1 Motivation

Problem-solving in Requirements Engineering 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.

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 Requirements Modelling Languages, 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 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 Chapter 9.

Secondly, committing already now to some specific categories would bias the discussion to a specific class of Problems. This is because Problem 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 Problem class over others this early in the tutorial.

Note that, while I am discussing relations before categories, I do not suggest, for example, that in general, relations should be the primitives in Requirements Modelling Languages. 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.

7.2 Single Relation Language

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 say 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.

Suppose that you need to define the simplest Requirements Modelling Language 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.
- 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?

7.2.1 Choose a Language Service

I will start by choosing the Language Service which I want the new language to deliver. To do this, recall that the default view in Requirements Engineering is that Problems 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 Problem).

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 London Ambulance, 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 Requirements Modelling Languages deliver. For example, in KAOS, the ability to answer "Which requirements are more detailed than (that is, refine) the given requirement?"; in i-star, to answer "Which tasks are more detailed than (decompose) the given task?". This leads me to the following Language Service for the new Requirements Modelling Language. Let x and y be parts of a model M in that language.

Language Service: AddsDetails

Does x add information to y in M? s.AddsDetails

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

7.2.2 Models over Fragments

Natural language text is an accessible and neutral way to represent information about Problems and their solutions. There is no need to learn something new to use it. 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 London Ambulance.

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

• FillIncRep: Fill out the incident report.
• FillSwIncRep: Fill out incident report form via software.

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*[1], not all of them are: questions arise during problem solving, and while they cannot be propositions [139, 53, 143] (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 [35, 120].

Consequently, *a Fragment is any available representation of information, as long as the model user judges it to be relevant for problem solving in Requirements Engineering.* This is important to keep in mind, as all languages in the rest of this book 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.

Fragments

Example 7.2.1. The London Ambulance 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.

[1] I take McGrath's view on propositions [104], 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."

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. •

Taken as a representation of information about ambulance dispatching, the paragraph in Example 7.2.1 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*.

7.2.3 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 7.2.1 and Example 7.2.2; the latter was made using L.D1 on Fragments in the former example.

Example 7.2.2. The graph $G = (X, \text{r.ifm})$ 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:

  r.ifm = {
  (RecEmCal, AddRepEm), (SwtchCal, AddRepEm),
  (NoDropCal, AddRepEm), (IdIncLoc, AddRepEm),
  (ChkDblLoc, AddRepEm), (FillIncRep, AddRepEm),
  (FillSwIncRep, FillIncRep) }

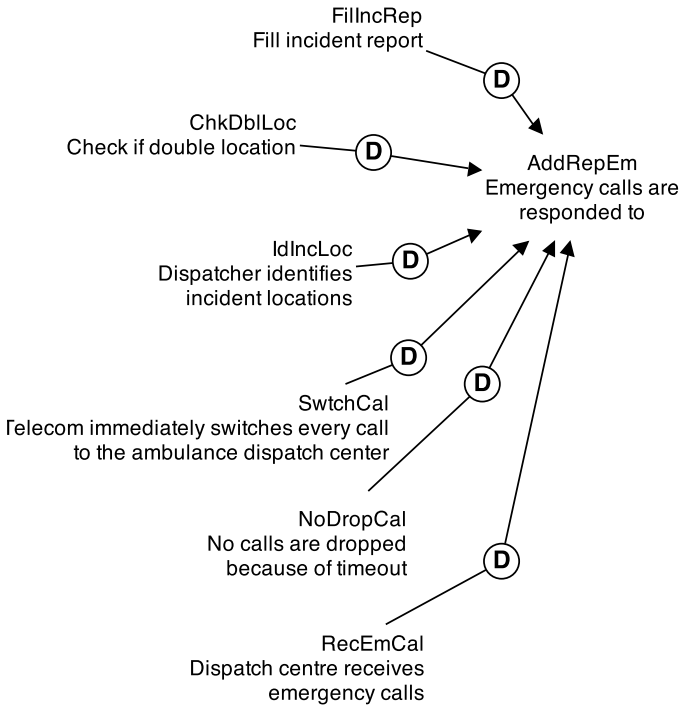


Figure 7.1: A visualisation of a model in L.D1.

Figure 7.1 gives a visualisation of this model. The visualisation shows a graph, where nodes are Fragments, and edges labeled "D" are r.ifm instances. •

The example illustrates why Language Services are interesting. Namely, *if I made the model in Example 7.2.2, 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.* Hence, L.D1 delivers s.AddsDetails.

As an aside, note that L.D1 cannot be used to solve the Default Problem. Delivering s.AddsDetails is not enough, as other Language Services are needed. If you consider that a language is not an Requirements Modelling Language if it cannot be used to solve the Default

Problem, then L.D1 is not one.[2] 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 book should start from something simple.

7.3 Modular Definitions

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.

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 do I want it to deliver?

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

Relation: ifm	
Inform	r.ifm
Domain & Dimension	

[2] i-star, for example, also fails this criterion, but is considered an Requirements Modelling Language. There is, to the best of my knowledge, no widely-accepted set of criteria for when a modelling language is also an Requirements Modelling Language, despite some suggestions [155, 64, 88, 83].

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.rel. 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 book.

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
r.ifm

L.Alpheratz

Domain

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

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{aligned} A &::= x \mid y \mid z \mid \ldots \\ B &::= A \text{ informs } A \\ Z &::= A \mid B \end{aligned}$$

Mapping

$\mathscr{D}(A) \in F$ and $\mathscr{D}(B) \in$ r.ifm, that is, every A symbol refers to a Fragment in **F** and every B symbol to an instance of r.ifm.

Language Services

Same as r.ifm.

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

I follow Harel & Rumpe [68] 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 book, but its definition is always local to a language. The example below gives a model in L.Alpheratz.

Example 7.3.1. The following is a model in L.Alpheratz:

M = { RecEmCal, SwtchCal, NoDropCal, IdIncLoc,
ChkDblLoc, FillIncRep, AddRepEm, FillSwIncRep,
RecEmCal informs AddRepEm,
SwtchCal informs AddRepEm,
NoDropCal informs AddRepEm,
IdIncLoc informs AddRepEm,
ChkDblLoc informs AddRepEm,
FillIncRep informs AddRepEm,
FillSwIncRep informs FillIncRep }

M includes individual Fragments, which I gave first above, and then 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 the following.

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

Suppose that $G(M)$ is the graph where every A symbol from M is a node and every B symbol $B = x$ informs 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 [2, 7]. They can be used to compute answers to the Language Services above.

In the rest of the book, 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

Given the relation r.R over Fragments in **F**, which graph is induced by that relation?

s.RelGraph

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

Function: map.abrel.g

Map a binary relation to a graph

f.map.abrel.g

Input

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

Do

Let G(X, r.R) = (N, E, l_N, l_E) be an empty labelled directed graph:

- For every Fragment $f_i \in X$, 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(X, r.R).

Language Services

- s.RelGraph: G(X, 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 book. 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
r.R, f.map.abrel.g

L.Alpheratz(R)

Domain

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

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$,

where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} A & ::= & x \mid y \mid z \mid \ldots \\ B & ::= & A \text{ symbol\_for\_R } A \\ Z & ::= & A \mid B \end{array}$$

Mapping

$\mathscr{D}(A) \in$ **F** and $\mathscr{D}(B) \in$ 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 7.4 and 7.5 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 Requirements Engineering, as well as relations that are central in well-known Requirements Modelling Languages. The aim is to give more examples of the modular definition of relations, and then combine these sample relations into new languages in Section 7.6.

7.4 Some Influence Relations

A recurrent concern in Requirements Engineering 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 Problem, 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 [122].

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.

7.4.1 Presence of Influence

If you want models to show only that influence does or should exist, then you need to deliver the following Language Service, and solve the exercise that follows.

Language Service: DoesInfluence

Does the satisfaction of a model part x influence the satisfaction of another model part y in the model M?

s.DoesInfluence

The Language Service can be delivered with a new relation which indicates influence.

Relation: inf

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 according to which the satisfaction value assigned to y depends on the satisfaction value assigned to x".

Language Services

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

Example 7.4.1. In Example 7.2.1, 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.inf) from Section 7.3, and r.inf. 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 7.2. For brevity, edges are labeled "I", rather than "r.inf". •

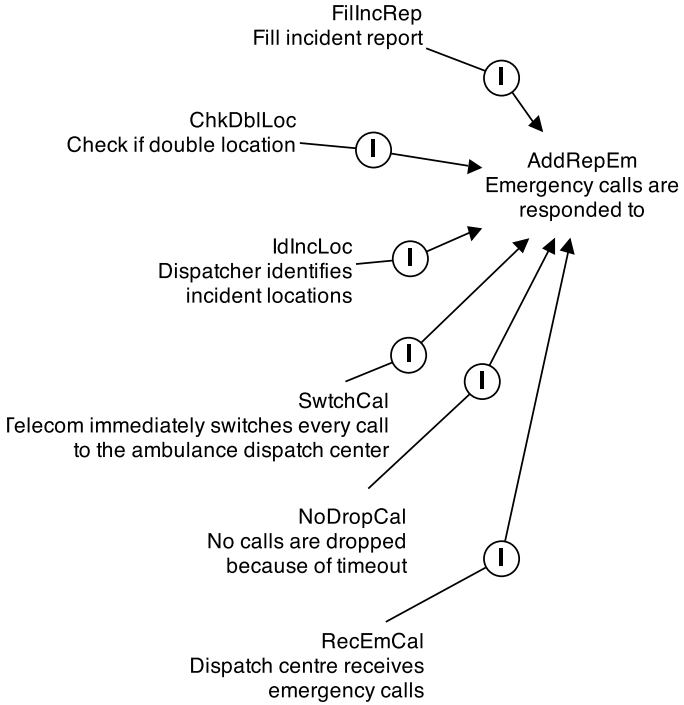


Figure 7.2: Visualisation of a model in L.Alpheratz(r.inf).

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.

7.4.2 Direction of Influence

You may not know exactly how the satisfaction of y depends on that of x, but 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.PosInfluence
- **s.NegInfluence**: Does satisfying x influence negatively the satisfaction of y in M? s.NegInfluence

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: inf.d
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".

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 7.4.2. 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
r.inf.pos, r.inf.neg, f.map.abrel.g

L.Ankaa

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

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$
\begin{array}{lcl}
A & ::= & x \mid y \mid z \mid \ldots \\
B & ::= & A \text{ influences+ } A \\
C & ::= & A \text{ influences- } A \\
Z & ::= & A \mid B \mid C
\end{array}
$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in$ **F**, B symbols denote r.inf.pos, and C symbols r.inf.neg instances, $\mathscr{D}(B) \in$ r.inf.pos and $\mathscr{D}C \in$ r.inf.neg.

Language Services

s.PosInfluence, s.NegInfluence.

Figure 7.3 shows a graph made by merging the graphs G(**F**, r.inf.pos) and G(**F**, r.inf.neg), both 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 $(x, y) \in$ 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.

7.4.3 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.

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?

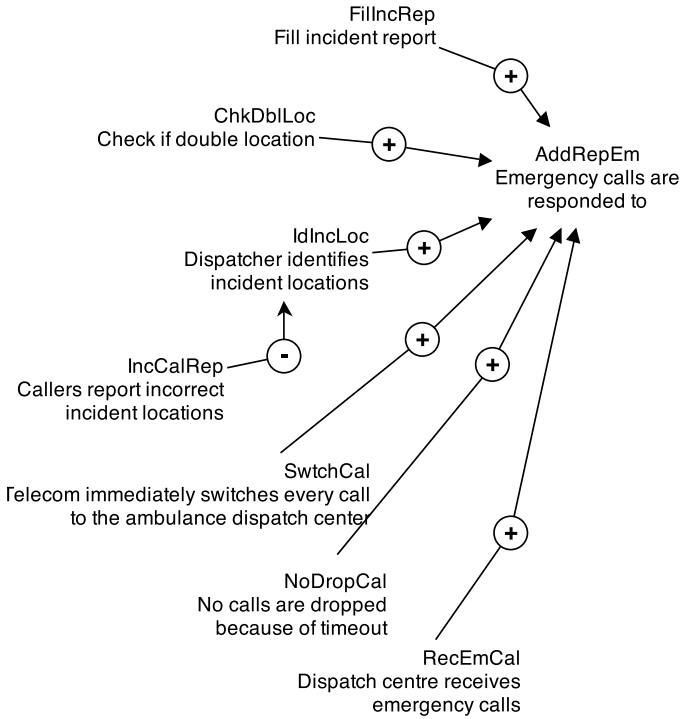


Figure 7.3: A visualisation of a model in L.Ankaa.

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

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: str.inf

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 value assigned to y is more sensitive to the satisfaction value assigned to x_i than to the satisfaction value assigned to x_j".

Language Services

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

Example 7.4.3. 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
r.inf.pos, r.inf.neg, f.map.abrel.g, r.str.inf

L.Schedar

Domain

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

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

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every Z is generated according to the following BNF rules:

$$
\begin{array}{lcl}
A & ::= & x \mid y \mid z \mid \ldots \\
B & ::= & A \text{ influences+ } A \\
C & ::= & A \text{ influences- } A \\
D & ::= & B \text{ infstronger } B \mid C \text{ infstronger } C \\
Z & ::= & A \mid B \mid C \mid D
\end{array}
$$

Mapping

$\mathscr{D}(A) \in F$, $\mathscr{D}(B) \in$ r.inf.pos, $\mathscr{D}(C) \in$ r.inf.neg, and $\mathscr{D}(D) \in$ r.str.inf.

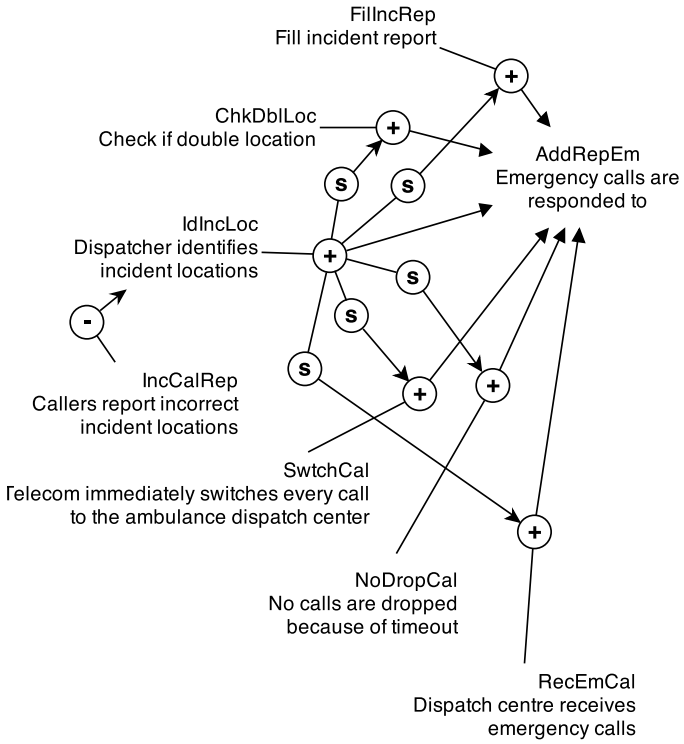


Figure 7.4: A visualisation of a model in L.Schedar.

Language Services

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

Figure 7.4 is a visualisation of a model 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 covari-

ance, 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 Problem instance that you are solving. For the former case, multivariate statistics [109, 138] 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 [125] is relevant.

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.

7.4.4 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.inf, and want to use r.inf.pos and r.inf.neg instances. As it further increases, I will want to use f.inf.str 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 Chapter 10.

7.5 Arguments in Models

A recurring concern in Requirements Engineering 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.

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 inspired by a central notion in program refinement.

Program refinement [151, 41, 73, 40] 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 book 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 would ideally remove, or more realistically 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 Requirements Engineering involves working with partial information. So it is often simply not feasible to provide conditions as clearly verifiable as proof obligations.[3]

[3]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

7.5.1 Support and Defeat

The obligation to have a justified model 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 *support* and *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:

- **s.DoesSupport**: Does accepting x support accepting y as well in M? s.DoesSupport
- **s.DoesDefeat**: Does accepting x support rejecting (not accepting) y in M? s.DoesDefeat

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

Relation: sup
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.

on this in Chapter 10).

Reading

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

Language Services

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

In contrast to r.sup, r.def is intransitive. This is an important property, and reflects 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: def

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.def reads "if x is accepted, then y should not be".

Language Services

- s.DoesDefeat: Yes, if there is $(x, y) \in$ r.def 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 7.5.1. 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

L.Diphda

Language Modules
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 $\subseteq \mathbf{F} \times \mathbf{F}$. A Fragment can act as a reason, or argument in favour or against a r.ifm instance or another Fragment, so that

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

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every Z is generated according to the following BNF rules:

$$\begin{aligned} A &::= x \mid y \mid z \mid \ldots \\ B &::= A \text{ informs } A \\ C &::= A \text{ supports } A \mid A \text{ supports } B \\ D &::= A \text{ defeats } A \mid A \text{ defeats } B \\ Z &::= A \mid B \mid C \mid D \end{aligned}$$

Mapping

A symbols denote Fragments, so $\mathscr{D}(A) \in \mathbf{F}$. B symbols denote r.ifm instances, $\mathscr{D}(B) \in$ r.ifm. C symbols and D symbols denote, respectively, instances of r.sup and 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 7.2.1, Fragments RecEmCal, SwtchCal, NoDropCal, IdIncLoc, ChkDblLoc, and FillIncRep described parts of what needs to be done, in order to satisfy AddRepEm. In Example 7.2.2, 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:

• HowSwtchCal: SwtchCal says how to satisfy AddRepEm.
• WhoSwtchCal: SwtchCal says who is involved in satisfying AddRepEm.
• WhenSwtchCal: SwtchCal says when some events happen when satisfying AddRepEm.
• HowNoDropCal: NoDropCal says how to satisfy AddRepEm.
• HowRecEmCal: RecEmCal says how to satisfy AddRepEm.
• WhoRecEmCal: RecEmCal says who is involved in satisfying AddRepEm.
• HowIdIncLoc: IdIncLoc says how to satisfy AddRepEm.
• WhoIdIncLoc: IdIncLoc says who is involved in satisfying AddRepEm.
• HowChkDblLoc: ChkDblLoc says how to satisfy AddRepEm.

• HowFillIncRep: FillIncRep says how to satisfy AddRepEm.

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 7.5 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 7.5.1 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 7.5.2. James and Jill are modelling requirements for London Ambulance. James made the model discussed in Example 7.5.1, visualised in Figure 7.5. 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.

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

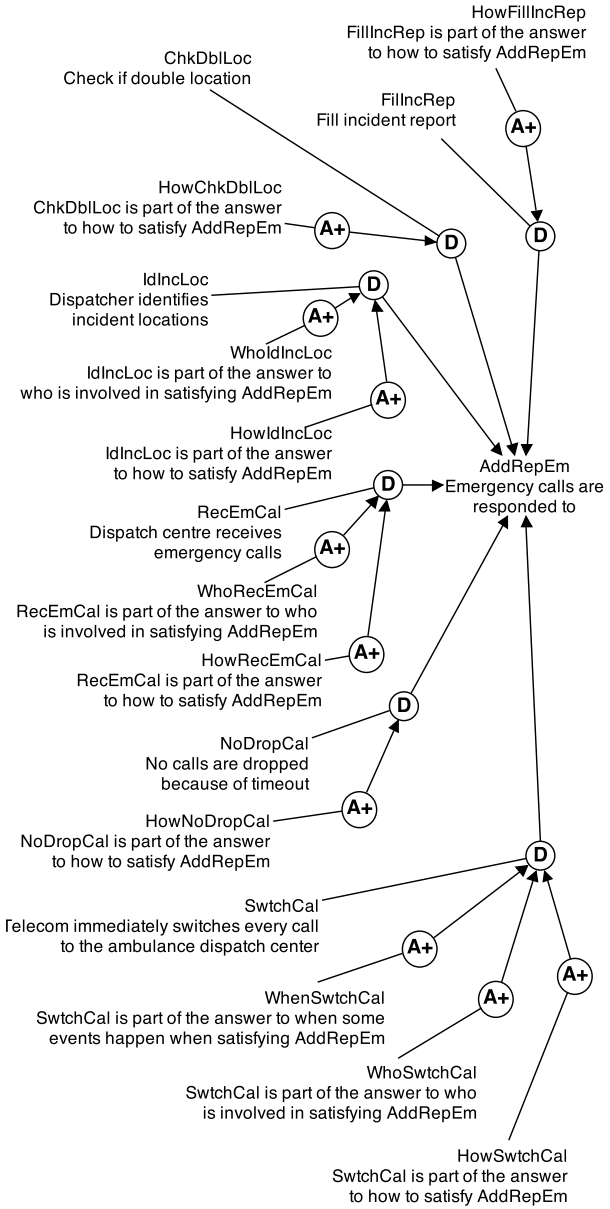


Figure 7.5: A visualisation of a model in L.Diphda.

And then, by adding the r.def instance

$$(\mathsf{RepFillOutScp}, \mathsf{HowFillIncRep})$$

I would update Figure 7.5 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 Requirements Engineering and elsewhere, which are used to represent the design rationale [94, 32, 96, 97, 117, 127, 126, 102, 90, 87, 82], 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 [121]. 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.

7.5.2 Accepted or Rejected

Having chains of r.sup and r.def instances raises the issue of *acceptability*. Acceptability is interesting, because something being acceptable is synonymous to it being justified. In Example 7.5.2, there was a chain made from $(\mathsf{RepFillOutScp}, \mathsf{HowFillIncRep}) \in \mathsf{r.def}$ and

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

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 (FillIncRep, AddRepEm) ∈ 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.

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 [147, 10, 72], and in artificial intelligence, in relation to argumentation systems [45, 27, 9] and defeasible logics [114, 129, 116].

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 [27] 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 7.5.3. 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 7.6(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.

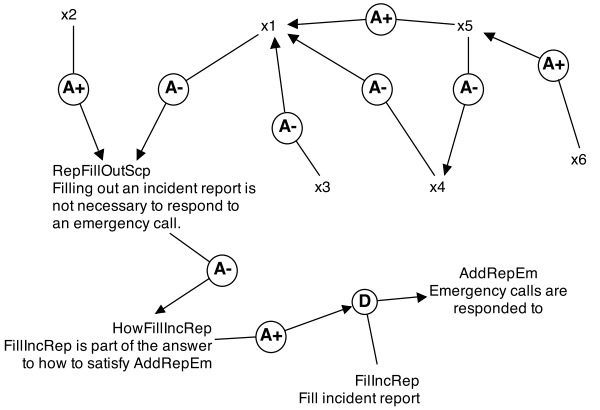
Which Fragments in Figure 7.6(a) are acceptable? Consider first the leaves, and observe that there are no arguments against AddRepEm, FillIncRep, x2, x3, x5 and x6, so that they are acceptable. x6 supports x5. Since x5 is acceptable and is against x4, x4 is not acceptable. Consequently, it does not matter 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.

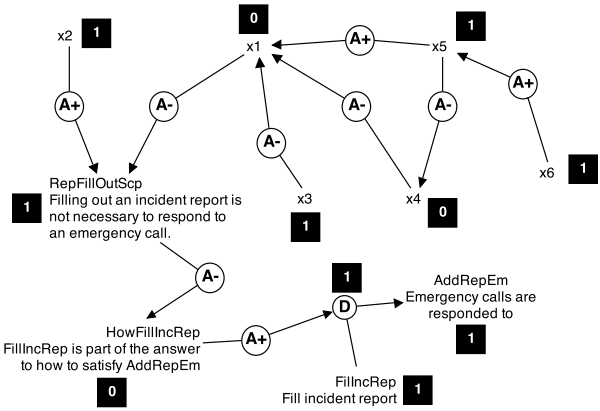
The resulting acceptability values are shown as additional markers on model elements in Figure 7.6(b). The model there has exactly one extension, and it includes all model parts which are marked with the acceptability value 1.

Figure 7.7(a) 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 [45]. This is convenient because it is simple and generalises many others in artificial intel-

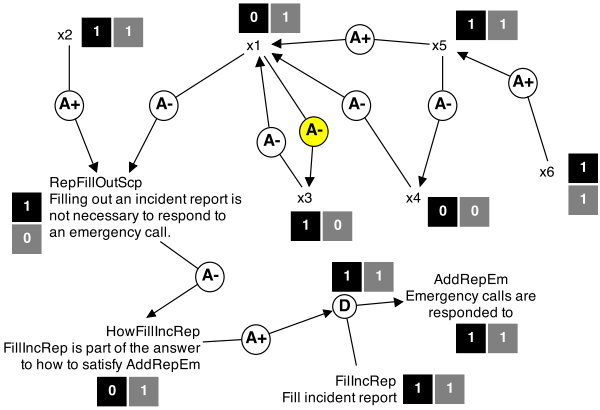


(a) A visualisation of a model discussed in Example 7.5.3.

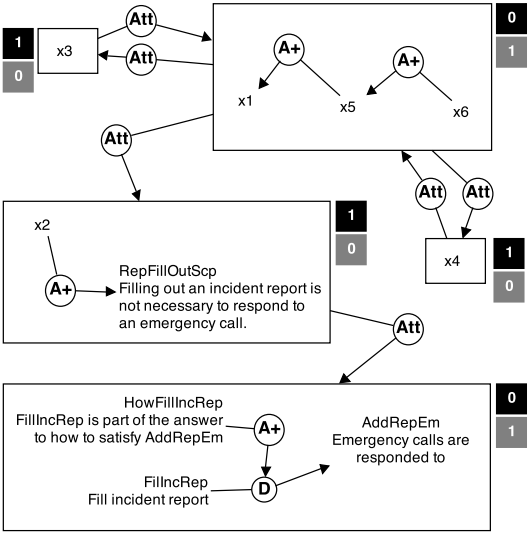


(b) Acceptability values for the model in Figure 7.6(a).

Figure 7.6: Illustration of acceptability values, part one.



(a) There are now two extensions.



(b) Dung argumentation framework from f.acc on Figure 7.7(a).

Figure 7.7: Illustration of acceptability values, part two.

ligence. The rough idea is similar (but not the same, as explained in Example 7.5.4) to that explained above with x, y, and z and in Example 7.5.3. 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?

s.IsAcceptable

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: acc

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 $l_{Arg}(a) = c$.
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 [105] 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 7.5.4. 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 7.7(a) gives the argumentation framework visualised in Figure 7.7(b). 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 7.7(a). 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 [74] and to compute extensions of Dung argumentation frameworks [105]. 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.[4]

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.[5] In con-

[4] 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 [45, 128].

[5] 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 [114]: "[...] inductive reasoning is not deductive, and in perception, when one judges the colour of something on

trast 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.

7.6 Combinations of 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 7.6.1 focuses on the simpler case of independence, and Section 7.6.2 on interaction.

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."

7.6.1 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.

7.6.2 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

L.Achernar

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

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$
\begin{array}{lcl}
A & ::= & x \mid y \mid z \mid \ldots \\
B & ::= & A \text{ informs } A \\
C & ::= & A \text{ influences+ } A \\
D & ::= & A \text{ influences- } A \\
Z & ::= & A \mid B \mid C \mid D
\end{array}
$$

Mapping

$\mathscr{D}(A) \in$ **F**, $\mathscr{D}(B) \in$ r.ifm, $\mathscr{D}(C) \in$ r.inf.pos, and $\mathscr{D}(D) \in$ 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.

$$
\begin{array}{lcl}
(x, y) & \in & \text{r.inf.pos} \\
(x, y) & \in & \text{r.inf.neg}
\end{array}
$$

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.

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?

s.NegWins

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.

7.7 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 Requirements Modelling Languages, and some of them will be discussed in the next sections. Chapter 8 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. Chapter 9 introduces categories, and illustrates how relations can be restricted to specific Fragment categories, which can reduce the number of relation interactions. Chapter 15 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. Chapter 12 uses n-ary relations in order to represent alternatives in models.

Chapter 8

Guidelines

This Chapter is on how to define guidelines for problem solving in Requirements Modelling Languages. Guidelines recommend how to do something in problem solving, so as to move closer to a solution. The Chapter focuses on the following questions.

1. *How to find guidelines for problem-solving, and embed them in Requirements Modelling Languages? (Section 8.2)*
2. *How to combine guidelines into new, more complicated ones? (Section 8.3)*
3. *How to strengthen or weaken guidelines, and why? (Section 8.4)*

8.1 Motivation

Guidelines suggest how to do problem solving in Requirements Engineering. 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.acc. 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 (or a relation which delivers the same Language Services as 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.

Requirements Modelling Languages 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 8.2 and 8.3 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 8.4 illustrates the ideas of strengthening and weakening guidelines and why that may be relevant.

8.2 Guidelines from 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.HowInforms**: 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.HowInforms
- **s.WhyInforms**: Given two Fragments x and y such that $(y, x) \in$ r.ifm, why does y add details to x? s.WhyInforms

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

If I want guidelines that are independent from the specific domain or Problem 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 7.5.1. 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 London Ambulance in Example 7.5.1, 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.ifm

Add details f.add.ifm

Input

$x \in$ **F**.

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 $\mathbf{F}_q$ and R_q, for

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

such that:

(a) each $y \in \mathbf{F}_q$ 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 $\mathbf{F}_q$ and R_q, for $q \in \{Who, How, When, Where, WhoFor\}$.

Language Services

- s.HowInforms: Apply f.add.ifm to Fragments in a given model M, and add the resulting sets back to M.
- s.WhyInforms: 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 $\mathbf{F}_q$. When a Fragment $y \in \mathbf{F}_q$ 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: q

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

Example 8.2.1. 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 L.Hamal
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 $M = \{Z_1, \ldots, Z_n\}$, where every Z is generated according to the following BNF

rules:

$$
\begin{array}{lcl}
A & ::= & x \mid y \mid z \mid \ldots \\
B & ::= & A \text{ informs } A \\
C & ::= & \mathit{Who} \mid \mathit{How} \mid \mathit{When} \mid \mathit{Where} \mid \mathit{WhoFor} \\
D & ::= & B \text{ answers } C \\
Z & ::= & A \mid B \mid D
\end{array}
$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$. B are for r.ifm instances, that is, $\mathscr{D}(B) \in$ r.ifm. D symbols denote r.q instances,

$$\mathscr{D}(B \text{ answers } \mathit{Who}) \in \text{r.Who}, \ldots,$$
$$\mathscr{D}(B \text{ answers } \mathit{WhoFor}) \in \text{r.WhoFor}.$$

Language Services

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

Figure 8.1 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 7.2.1. •

8.3 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 some influences that of others, are closely related to

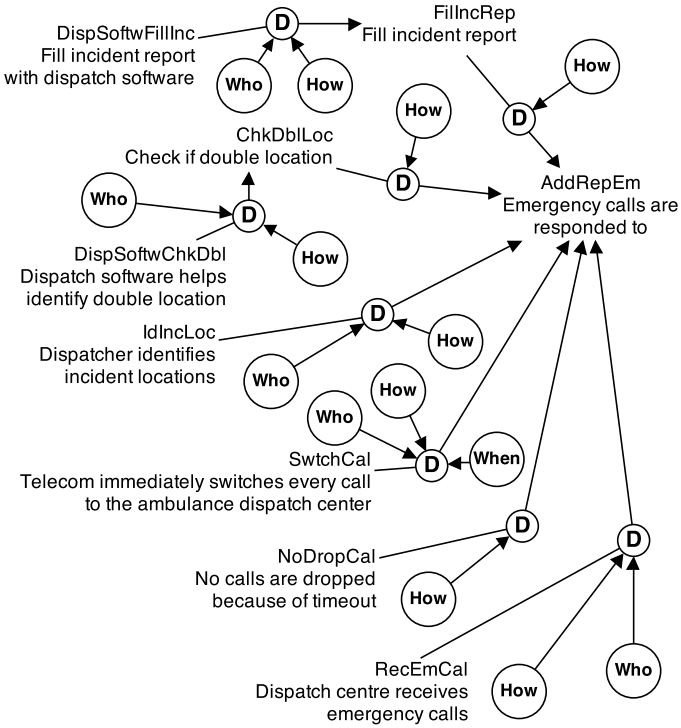


Figure 8.1: A visualisation of a model in L.Hamal.

the issue of operationalisation in Requirements Engineering. 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.

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?	s.AreOpr

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

Function: opr.all	
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 8.3.1. 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 L.Acamar

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

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every Z is generated according to the following BNF rules:

$$
\begin{array}{lcl}
A & ::= & x \mid y \mid z \mid \ldots \\
B & ::= & A \text{ informs } A \\
C & ::= & A \text{ influences+ } A \\
D & ::= & A \text{ influences- } A \\
Z & ::= & A \mid B \mid C \mid D
\end{array}
$$

Mapping

$\mathscr{D}(A) \in$ **F**, $\mathscr{D}(B) \in$ r.ifm, $\mathscr{D}(C) \in$ r.inf.pos, and $\mathscr{D}(D) \in$ r.inf.neg.

Language Services

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

Figure 8.2 is a visualisation of a model in L.Acamar, made by applying f.opr.all to the Fragment AddRepEm. •

8.4 Stronger and Weaker 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

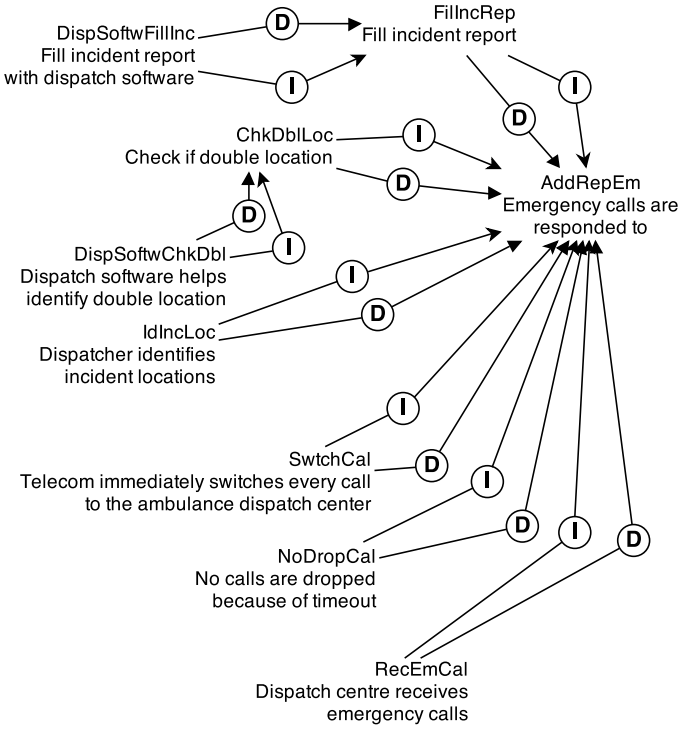


Figure 8.2: A visualisation of a model in L.Acamar.

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 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.

8.5 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.

Chapter 9

Categories

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

1. *Why and how to use independent categories? (Section 9.2)*
2. *What to do when there is a taxonomy of categories? (Section 9.3)*
3. *What is the meta-model, and what the ontology of a language? (Section 9.4)*
4. *Why and how to define derived categories and relations in a language? (Section 9.5)*
5. *How to enforce the intended use of categories in a language? (Section 9.6)*

9.1 Motivation

A category groups Fragments which share the same properties. Categories are used to distinguish between Fragments having different properties, and thereby should be used differently during problem-solving.

In absence of categories, it is not possible, for example, to make a language which represents instances of the Default Problem. This is because the Default Problem 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 Problem instances.

9.2 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 Problem 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 these categories are defined [155]:

> *"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."

Given the quote above, consider how you would 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?

I define each of the three categories with a Language Module. The Language Module has the same slots as for relations, which is unsurprising, since you can see categories as unary relations. But it should be clear when I am talking about relations, and when about categories, and consequently categories have their own Language Module. Below is a definition of the requirement category, inspired by the definition of requirement in Default Problem.

Category: r	
Requirement	c.r
Domain c.r $\subseteq$ **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" [155].	
Reading $x \in$ c.r reads "x is a requirement".	
Language Services	

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

The "membership conditions" 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: k

Domain knowledge c.k

Domain

c.k $\subseteq$ **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" [155].

Reading

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

Language Services

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

And finally, there is a category for specifications.

Category: s

Specification c.s

Domain

c.s ⊆ **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. s.IsSpec

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

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 Problem, 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: cat.ksr

Categorise in Default Problem 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 9.2.1. 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 L.Menkar
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, $\mathbf{F} = \mathsf{c.r} \cup \mathsf{c.k} \cup \mathsf{c.s}$ and $\mathsf{c.r} \cap \mathsf{c.k} \cap \mathsf{c.s} = \emptyset$. Influence relations are over Fragments, $\mathsf{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\mathsf{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$. 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 $M = \{Z_1, \ldots, Z_n\}$, where every Z is generated according to the following BNF rules:

$$\begin{array}{lcl} A & ::= & x \mid y \mid z \mid \ldots \\ B & ::= & r(A) \mid k(A) \mid s(A) \\ C & ::= & B \text{ influences+ } B \\ D & ::= & B \text{ influences- } B \\ E & ::= & C \text{ infstronger } C \mid D \text{ infstronger } D \\ Z & ::= & B \mid C \mid D \mid E \end{array}$$

Mapping

A symbols denote uncategorised Fragments. B symbols denote categorised Fragments, so $\mathscr{D}(r(A)) \in \mathsf{c.r}$, $\mathscr{D}(k(A)) \in \mathsf{c.k}$, and $\mathscr{D}(s(A)) \in \mathsf{c.s}$. C symbols denote positive influence relations, $\mathscr{D}(C) \in \mathsf{r.inf.pos}$, and D negative influence relations, $\mathscr{D}(D) \in \mathsf{r.inf.neg}$. E symbols denote relative strength of influence, $\mathscr{D}(E) \in \mathsf{r.str.inf}$.

Language Services

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

Figure 9.1 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. •

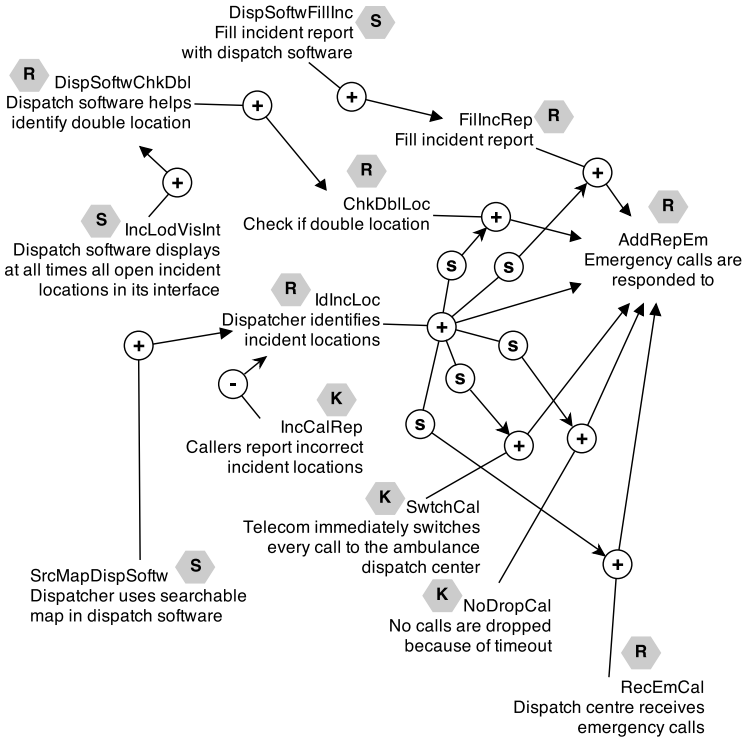


Figure 9.1: A visualisation of a model in L.Menkar.

9.3 Taxonomy of Categories

A taxonomy of categories is a set of categories related by the is-a relation, also called the specialisation 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. In more technical terms, the extension of category A is a subset of the extension of B.

It is common in Requirements Engineering to distinguish between two kinds of requirements, often called functional and nonfunctional requirements. I can have two categories for them, both specialisations of c.r.

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. This follows oft-cited research in Requirements Engineering, such as the NFR framework. According to this view, being able to communicate via radio with an ambulance is a functional requirement, while quickly responding to incidents is a nonfunctional requirement.

Category: r.f
Functional requirement c.r.f
Domain c.r.f $\subseteq X$, 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. — s.IsFunctReq

Category: r.nf	
Nonfunctional requirement	c.r.nf
Domain c.r.nf $\subseteq X$, 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.	s.IsNFunctReq

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 9.2.

If a category is a specialisation of another one, then the former inherits the properties of the latter. Modules above captured inher-

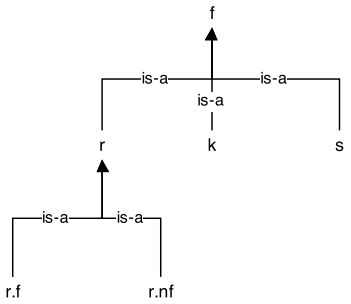


Figure 9.2: Taxonomy of categories from Sections 9.2 and 9.3.

itance 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 9.2 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.

9.4 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 Requirements Modelling Languages, 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 Problems [88]. In *formal ontology*, such a specification is written in a formal logic [62, 132, 134].

The meta-model and ontology of a language should not be confused [39]. 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 book, 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 Requirements Engineering. 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-star, KAOS, Techne, NFR, among others.

One way, then, to think of the ontology of an Requirements Modelling Language, is that it is the definition of the categories and relations needed to define instances of the requirements problem which the language is made 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 is to see the ontology of the language only as all categories and relations of the language. In Requirements Modelling Languages, 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 9.4.1. Figure 9.3(a) shows the categories and relations of two different languages, in Figures 9.3(a) and 9.3(b). Nodes represent categories and links represent relations.

Figure 9.3(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 9.2.

Figure 9.3(b) shows an ontology with same categories as in Figure 9.3(a), but now, the influence relations can go only from specification Fragments to requirement Fragments.

•

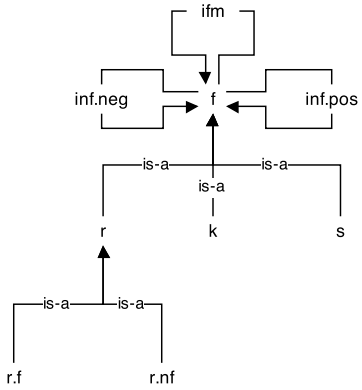
9.5 Derived Categories and Relations

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.

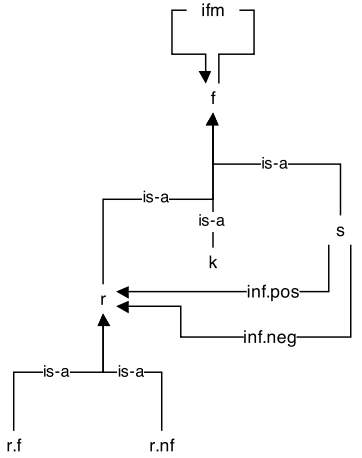
Derived categories and relations can be used to emphasise specific ideas in a language, or, for example, to simplify modelling. They are syntactic sugar in an Requirements Modelling Language. The following example illustrates this.

Example 9.5.1. Figure 9.3 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: rls	
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	



(a) Visualisation of the ontology of one language.



(b) Visualisation of the ontology of another language.

Figure 9.3: Visualisation of two ontologies.

$(x, y) \in$ r.rls reads "specification x realises the requirement y".

Language Services

Inherits from r.inf.

r.els is the abbreviation of an influence from a specification to a requirement. You may want to distinguish r.rls from others in an Requirements Modelling Language, 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 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 9.5.2. 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: r.clsh

Clashing requirement c.r.clsh

Domain

c.r.clsh $\subseteq X$, 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. s.IsClshReq

Example 9.5.3. Now suppose that I want to categorise a requirement as irrelevant, if that requirement is not acceptable. Acceptability works as in Section 7.5.2. I use f.Accepted to define the category c.r.irrl. •

Category: r.irrl

Irrelevant requirement c.r.irrl

Domain

c.r.irrl $\subseteq X$, 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. s.IsIrrlReq

9.6 Enforce Category Use

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 9.6.1. 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: chk.rop

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.

s.IsROpComp

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

9.7 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 (to be in the extension of) the category C?
  2. Can members of the extension of C be members of the extensions 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? This is answered by the relations over members of extensions, of the 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.

Chapter 10

Valuation

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, Value Assignment, and Outcome are central notions in valuation. A Value Type is simply a set of values, such that a variable x has Value Type T, if and only if any value of that variable must be a member of T. A Value Assignment is the value that a variable has, among the values of its Value Type. An Outcome is a non-empty set of Value Assignments. The Chapter looks at how to define Value Types, how to compute Value Assignments and Outcomes, and how to use all of these 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 10.2),*

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 10.3),*

3. *How to, and why define a language with an unordered set of values as its only Value Type? (Section 10.4),*

4. *What if the value type for the language is an ordered set? (Section 10.5),*

5. *Why and how to have in a language, a Value Type defined over real numbers? (Section 10.6).*

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

10.1 Motivation

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 book, 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 Chapter will illustrate how to answer the questions above for various Value Types.

10.2 Propagating Binary Values

This section discusses and combines two 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 10.2.1)
- 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 10.2.2)

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 which I refer to as "value propagation". In this approach, 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.

10.2.1 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?	s.SatReq

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.

10.2.2 Value Propagation

The language needs to represent if 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 later below.

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 will add a new function called f.sat.inf.pos. It relies on f.sat to return the satisfaction value of a Fragment. f.sat remains undefined for the moment. When I define it later,

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".

Value Assignment notation

Function: sat.inf.pos
Positive influence satisfaction
Input $(y, x) \in$ r.inf.pos and model M.
Do $v = 1$ if y is satisfied in M, and $v = 0$ otherwise.
Output $\langle (y, x)$, v.Satisfaction, $v \rangle$.
Language Services • **s.WhPosInfSat**: What is the v.Satisfaction value of $(y, x) \in$ r.inf.pos? : $\langle (y, x)$, v.Satisfaction, $v \rangle$.

f.sat.inf.pos

s.WhPosInfSat

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.

Value propagation

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 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: sat.inf.neg

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? : $\langle (y, x), \text{v.Satisfaction}, v \rangle$. s.WhNegInfSat

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, it will be manually assigned. I therefore 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]$(w,x) = 0$?

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, defined as follows.

Function: sat

Satisfaction f.sat

Input

Fragment $x \in$ **F** 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,

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

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. f.sat.inf.neg$((q_j,x),M)$ returns the satisfaction value of $(q_j,x) \in$ r.inf.neg in M.

Output

⟨x, v.Satisfaction, v⟩

Language Services

- **s.WhSat**: What is the satisfaction value of x in M? : It is ⟨x, v.Satisfaction, v⟩. s.WhSat

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.

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: sat.leaf

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. s.WhAsmSatLf

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 L.Rigel
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

Domain

There is a set of Fragments **F**, a singleton for Value Types

$$\mathbf{T} = \{\text{v.Satisfaction}\},$$

and a set of Value Assignments **V**. 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

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

Satisfaction is binary, v.Satisfaction = {1, 0}.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} A & ::= & x \mid y \mid z \mid \ldots \\ B & ::= & r(A) \mid k(A) \mid s(A) \\ C & ::= & B \text{ influences+ } B \\ D & ::= & B \text{ influences- } B \\ G & ::= & \langle A, E, F \rangle \\ Z & ::= & B \mid C \mid D \mid G \end{array}$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$. B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. C and D symbols denote, respectively, positive and negative influence relations. E symbols denote Value Types, $\mathscr{D}(E) \in \mathbf{T}$. F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction. G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$.

Language Services

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

Models in L.Rigel can represent mutually exclusive Value Assignments on same Fragments and relation instances, and therefore, mutually exclusive Outcomes 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 14). This is illustrated in Example 10.2.1 below.

In Example 10.2.1, Figures 10.1 and 10.2 do not include Outcomes. Figure 10.3 includes one Outcome, and Figure 10.4 includes two. An Outcome can be specific to one Value Type, as in Figures 10.3 and 10.4, 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 10.3 and 10.4 show v.Satisfaction Outcomes.

Example 10.2.1. This example illustrates how L.Rigel computes Value Assignments in a model. Figures 10.1–10.4 show four models in L.Rigel.

The first model in Figure 10.1 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 10.2 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 10.1.

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 10.4 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,

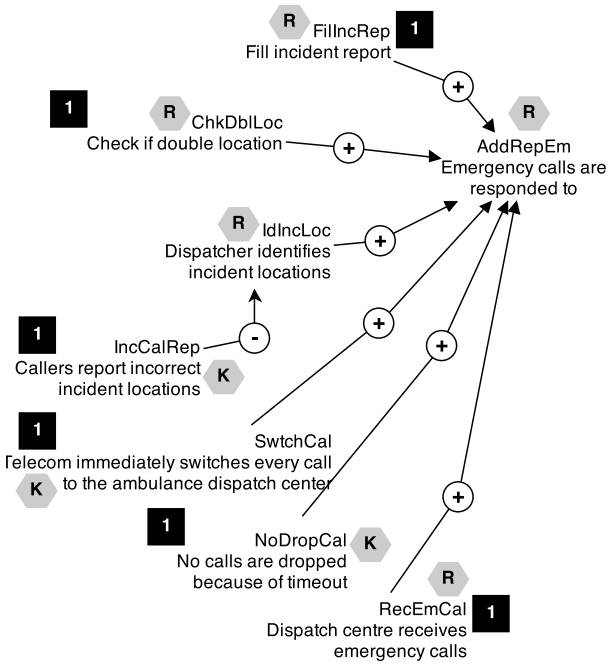


Figure 10.1: Satisfaction values assigned with f.sat.leaf.

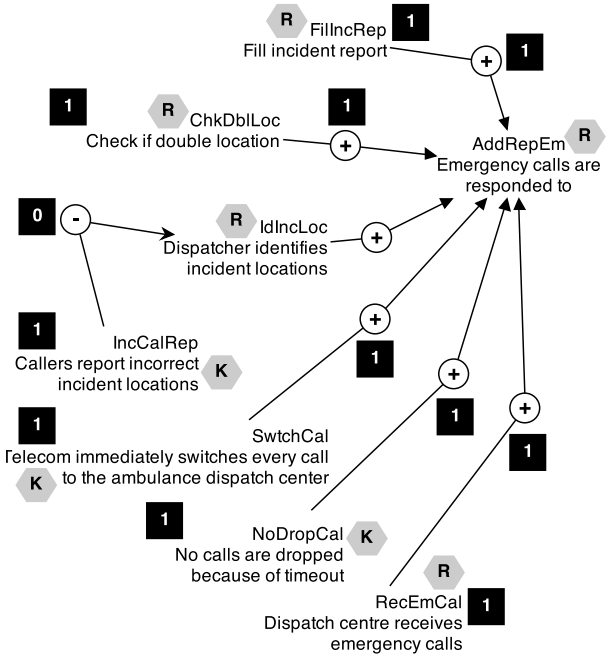


Figure 10.2: After applying f.sat.inf.pos and f.sat.inf.neg.

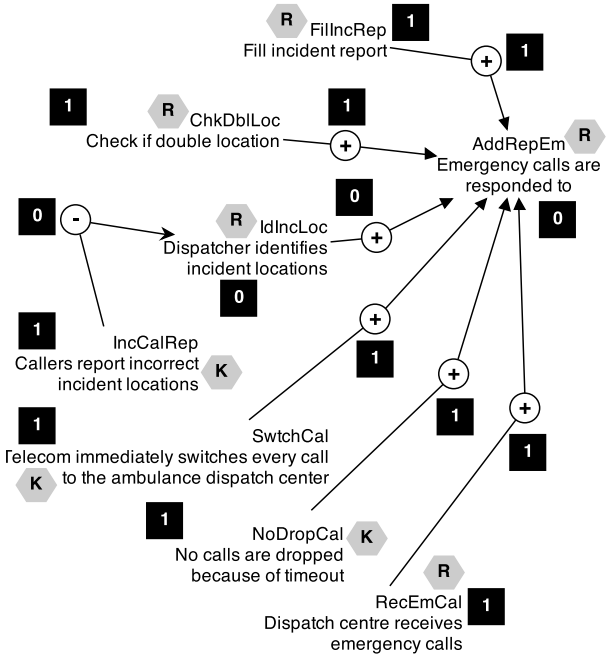


Figure 10.3: One Outcome.

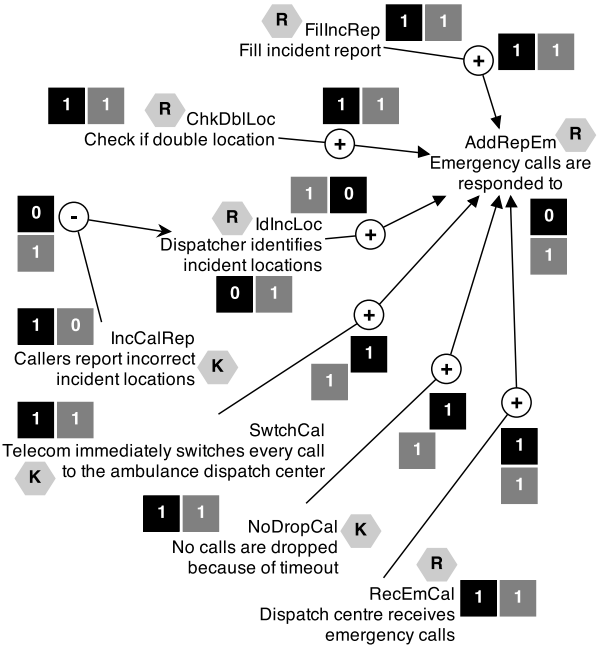


Figure 10.4: Two Outcomes.

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.

10.3 Combining 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? s.AppSat

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 10.2, 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 10.2. 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 10.3.1 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 10.3.2 focuses on the case where missing approval values can be computed from those that exist in a model.

10.3.1 Independent Value Assignments

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?

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: app.asg.ind

Assume independent approval value f.app.asg.ind

Input

Fragment $x \in \mathbf{F}$.

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. s.AsmApp

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 [33] and social choice [28, 5] 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 10.1 summarises these ideas.

Table 10.1: 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 10.3.1. 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: app.maj	
Majority approval	f.app.maj
Input	

Fragment $x \in \mathbf{F}$.

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.

s.IsMajApp

Example 10.3.2. 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: sat.nxt

Satisfy next

f.sat.nxt

Input

Fragment $x \in \mathbf{F}$.

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 $\langle x, \text{v.SatNext}, 1\rangle$. s.DoSatNext

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 10.4.

10.3.2 Dependent Value Assignments

In Section 10.3.1, 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.

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 10.2. 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?

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. f.sat answers the third question above. Finally, f.sat.leaf was the answer to the fourth question.

10.4 Sets of Values

Consider now a Value Type which is a set of values, and there is no order over them. In Section 10.3, there were four combinations of binary values from two core binary Value Types, v.Satisfaction and v.Approval. Table 10.1 gave a reading of these combinations. The four combinations can be used to define the three values of a new Value Type, 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.

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?

s.WhichDetail

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: ask.next

What to ask next

f.ask.next

Input

Fragment $x \in$ **F** 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 missing from V, where V is from $\langle x, \text{v.AskNext}, V \rangle$.

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.

10.5 Constraints on Assignments

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?

s.WorkProgrRep

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.

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: chk.progrstatus

Check progress status sequence f.chk.progrstatus

Input

Fragment $x \in$ **F** 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.

10.6 Real Numbers

The hypothetical work process in Section 10.5 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.

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 you 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, which uses the language and is designing the system, 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: chk.20more

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. s.WhAppAgain

10.7 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. I focused on functions which values of binary Value Types in models. This showed one compelling

reason for having Value Types in the first place, and thinking about valuation in a language.

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 Chapter 11.
- How to say that some Value Assignments are mutually exclusive, and thereby enable a language to represent alternative Problem and solution instances? This is discussed in Chapter 12.
- 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 Chapter 14.
- 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 Chapter 15.

Chapter 11

Uncertainty

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 Chapter 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 11.2),*
- *What to do when there are dependent random variables in a model? (Section 11.3).*

11.1 Motivation

In Section 10.6, 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. I could not deliver the following Language Service, for example.

Language Service: UncImplTime

How uncertain is the assignment of the v.ImplTime value to the Fragment x in M?

s.UncImplTime

If a language can deliver s.UncImplTime, then its models can also answer such questions as, for example, "How uncertain is it that the implementation time of x will be v?", where v is the v.ImplTime value assigned to x.

This 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 show more realistic requirements in a model. To avoid this assumption, 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?

s.UncSat

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? s.UncApp

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.

11.2 Independent Random Variables

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 Probability space

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 book, the same as wanting the value of $P(\langle x, \text{v.Satisfaction}, 1\rangle)$.

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

$$P[a \leq v_x \leq b] = \int_a^b pdf(v_x)\,dv.$$

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 $P[x.\text{v.ImplTime} \leq 13]$. 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 $pdf(x.\text{v.ImplTime})$ 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 $pdf(x.\text{v.ImplTime})$, 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. It can be defined as follows.

Function: prob.asg

Assign probability value f.prob.asg

Input

- Assignment either of a single value $\langle x, \text{v.T}, w\rangle$ or of a range $\langle x, \text{v.T}, w_1 \le v \le w_2\rangle$ to random variable of Value Type T on Fragment x, and
- Value Type v.prob(x.v.T), defined by the probability space $(\mathcal{S}, \mathcal{E}, P)$ 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 $\langle x, \text{v.T}, w_1 \le v \le w_2\rangle$, then $p = P[w_1 \le v_x \le w_2]$.

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? $\langle x, \text{v.prob}(x.\text{v.T}), p\rangle$ which f.prob.asg returns.

s.WhProbability

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, because the required data may be hard to find, and there can be biases [141]. 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 [21] 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

$$\text{v.prob}(x.\text{v.Satisfaction})$$

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 11.2.1. 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: prob.sat.ind

Compute probability that a Fragment is satisfied using probabilities of satisfaction of Fragments which influence it positively

f.prob.sat.ind

Input

Fragment $x \in \mathbf{F}$ 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

$$y_1.\text{v.Satisfaction}, \ldots, y_n.\text{v.Satisfaction},$$

   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? : $\langle x, \text{v.prob}(x.\text{v.Satisfaction}), P(x.\text{v.Satisfaction} = 1)\rangle$.

s.WhProbSatInd

The language below allows random variables in models, and has f.prob.asg and f.prob.sat.ind.

Language: Adhara

Language Modules
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, f.prob.asg, f.prob.sat.ind

L.Adhara

Domain

There is a set of Fragments **F**, a singleton for Value Types

$$\mathbf{T} = \{\text{v.Satisfaction}\},$$

and a set of Value Assignments **V**. Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \text{c.r} \cup \text{c.k} \cup \text{c.s}$ and $\text{c.r} \cap \text{c.k} \cap \text{c.s} = \emptyset$. Influences are over Fragments, $\text{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\text{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$. Value Assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

$$\begin{aligned}\mathbf{V} \subseteq\ & W \times \{\text{v.Satisfaction}\} \times \text{v.Satisfaction} \\ & \cup\ W \times \{\text{v.prob}(x.\text{v.Satisfaction}) \mid xW\} \times [0,1], \\ & \text{where } W = \mathbf{F} \cup \text{r.inf.pos} \cup \text{r.inf.neg}.\end{aligned}$$

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,

$$\mathbf{T} = \{\text{v.Satisfaction}\} \cup \{\text{v.prob}(x.\text{v.Satisfaction}) \mid x \in W\},$$
$$\text{where } W = \mathbf{F} \cup \text{r.inf.pos} \cup \text{r.inf.neg}.$$

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 $M = \{Z_1, \ldots, Z_n\}$, where every Z is generated according to the following BNF rules:

$$\begin{aligned} A &::= x \mid y \mid z \mid \ldots \\ B &::= r(A) \mid k(A) \mid s(A) \\ C &::= B \text{ influences+ } B \\ D &::= B \text{ influences- } B \\ G &::= \langle A, E, F \rangle \\ Z &::= B \mid C \mid D \mid G \end{aligned}$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$. B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. C and D symbols denote, respectively, positive and negative influence relations. E symbols denote Value Types, $\mathscr{D}(E) \in \mathbf{T}$. F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction. G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$.

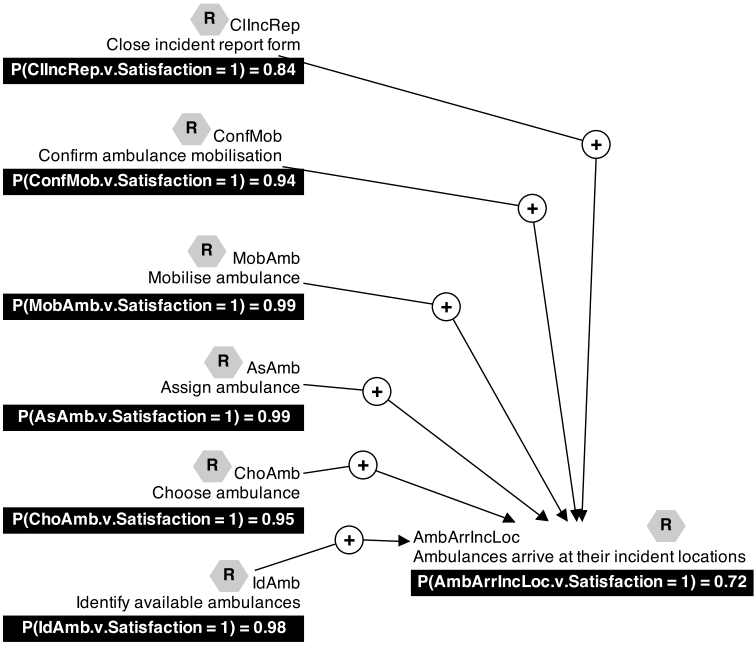


Figure 11.1: A model in L.Adhara with assignments of probability values.

Language Services

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

Figure 11.1 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. •

11.3 Dependent Random Variables

This section drops two assumptions made in Section 11.2: (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.

A language can use Bayesian networks [111, 24] to represent dependency between 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 11.3.1 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

11.3.2 automatically generates a Bayesian network, based on a graph in a model of the language. I will consider both options below.

11.3.1 Ignoring Existing Relations

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.baynet

Make a Bayesian Network f.make.baynet

Input

Set $X \subset \mathbf{F}$ 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).

s.WhProbBN

11.3.2 Using Existing Relations

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: map.inf.pos.baynet

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 11.3.1. Figures 11.2 and 11.3 give a simple and hypothetical example of applying f.map.inf.pos.baynet to a graph G(X, r.inf.pos) in Figure 11.2.

Every Fragment in the graph G(X, r.inf.pos) in Figure 11.2 has an associated random variable of the format $v[\ldots]$ in Figure 11.3. Figure 11.3 shows a Bayesian network, where edges are marked "B",

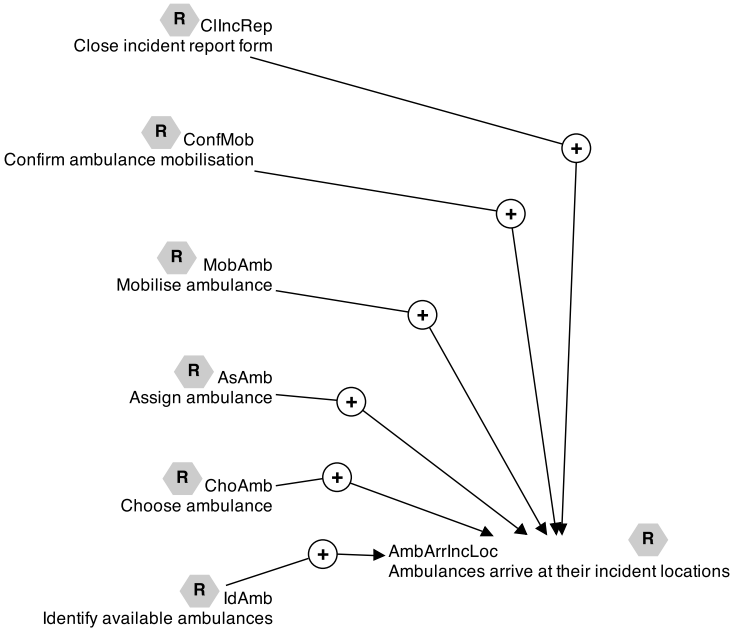


Figure 11.2: A model in L.Adhara, with no assignments of probability values.

made by applying f.map.inf.pos.baynet to the graph G(X, r.inf.pos) in Figure 11.2. Hypothetical probability values to root nodes, and the conditional probability values to the one non-root node were assigned manually. •

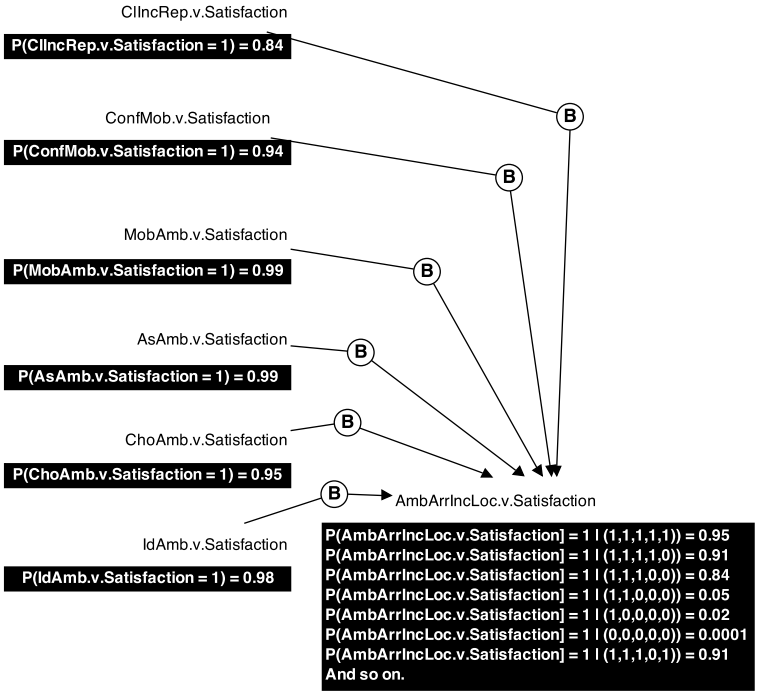


Figure 11.3: Bayesian network made by applying f.map.inf.pos.baynet to the model in Figure 11.2.

Chapter 12

Alternatives

This Chapter focuses on how to represent mutual exclusion in models. If parts A and B in a model are mutually exclusive, and that model represents one or more problem and, or solution instances, then none of these problems and solutions includes both A and B. In such a model, it may be that some problem (or solution) instances include A, others only B, some perhaps neither, but none will include both. I use two notions to discuss how mutual exclusion can work in languages in this book. One, called "Alternative", allows me to represent that, say, two Value Assignments are mutually exclusive. The other is Outcome, which was introduced in Chapter 10. Using these notions, I discuss the following questions.

1. *How to represent Alternatives? (Section 12.2)*
2. *How to find an Outcomes which include no mutually exclusive Value Assignment? (Section 12.3)*
3. *How to find Outcomes which include no mutually exclusive Value Assignment, when the language has several different Value Types? (Section 12.4)*

12.1 Motivation

The exact Problem instance to solve is often discovered during problem-solving. It is not known up front, but discovered and designed along the way. It is also rarely the case that you discover only one specific problem instance. There may be *different* sets of more concrete requirements, for example, such that each of these sets is an acceptable way to add details to the same less concrete requirement. You are thereby discovering a problem space, that is, a variety of problem instances, one or some of which your solution will solve. And the same applies to solutions, in that there is a solution space, rather than a single solution. The challenge is, then, to understand the problem space and the solution space, and find a pair, made of a problem instance and a solution instance, as the outcome of problem-solving.

This Chapter focuses on how to represent mutual exclusion in models. If parts A and B in a model are mutually exclusive, and that model represents one or more problem and, or solution instances, then none of these problems and solutions includes *both* A and B. In such a model, it may be that some problem (or solution) instances include A, others only B, some perhaps neither, but none will include both.

In languages in this book, Fragments or relation instances are not mutually exclusive themselves, but Value Assignments on them are. This is because saying that Fragments x and y are mutually exclusive is imprecise. Having Fragments be mutually exclusive may reflect that they should not be satisfied together, or that they cannot both be approved by stakeholders, or that they should not be included in the same release of the system together, and so on. Yet it could be that they both can be approved by stakeholders, but that they should not both be included in the same release of the system. The point being that, when mutual exclusion is about Value Assignment, then such ambiguities can be avoided.

I use two notions to discuss how mutual exclusion can work in languages in this book. One, called "Alternative", allows me to represent that, say, two Value Assignments are mutually exclusive. The other is Outcome, which was introduced in Chapter 10.

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.

To represent different ways of doing FillIncRep, and do so with languages defined so far, I would have to make one model each of these mutually exclusive ways. This is impractical. Suppose that there are three different ways to fill out a report, and two ways to allocate responsibility for doing so. This gives eight mutually exclusive Outcomes, and they cover only some options and only for FillIncRep, not other Fragments. Moreover, if the Requirements Modelling Language cannot represent all of them, it will not be able to represent relations between these Outcomes. You could thus have many models, but have no information *in these models* about which of them 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 only one. The concept of Alternative is a basic notion in decision making. 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.

Alternative

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. You may want to postpone choosing an Alternative, because you expect that there may be others which might also be worth considering. Or you may not have the authority to choose Alternatives yourself, but need to present them to stakeholders who have the authority to decide. Perhaps you also want first to find criteria for the comparison of the Alternatives (more on this in Chapter 14), before doing anything else with them.

12.2 Alternatives over Binary Value Types

Suppose that you want a language to deliver the following Language Service.

Language Service: SatAlt

Which are all the different ways for satisfying x according to the model M? s.SatAlt

This section focuses on the simpler case, where v.Satisfaction is binary.

Recall that L.Rigel can show positive and negative influence relations over Fragments, and propagate binary satisfaction values. But it cannot show that some Value Assignments are mutually exclusive.

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

A Value Assignment becomes an Alternative to another Value Assignment if it participates in a relation. I will call this relation vr.alt.b when it is over Value Assignments of binary Value Types. It is a binary relation over Value Assignments. It should be irreflexive, so that I cannot write that a Value Assignment is mutually exclusive to itself. It should also be symmetric, because if Value Assignments v and w are mutually exclusive, then each is mutually exclusive to the other. Finally, it is intransitive, as saying that v is mutually exclusive to w, and w to q does not necessarily mean that v and q are mutually exclusive.

Relation: alt.b

Mutually exclusive Value Assignments of a binary Value Type vr.alt.b

Domain & Dimension

r.alt.b $\subseteq$ **V** $\times$ **V**, where **V** is a set of Value Assignments of the same binary Value Type.

Properties

- Irreflexive, symmetric, and intransitive.
- If $(v, w) \in$ vr.alt.b in a model M, then there is no Outcome of M which includes both v and w.

Reading

$(v, w) \in$ vr.alt.b reads "Value Assignments v and w are Alternatives".

Language Services

- **s.IsAlt**: Are Value Assignments v and w Alternatives? Yes, if $(v, w) \in$ vr.alt.b. s.IsAlt

I define below the language L.Mirfak, which can represent positive and negative influence over requirements, domain knowledge, and specifications, just as L.Rigel. L.Rigel, however, cannot represent Alternatives. But because of Alternatives, L.Mirfak cannot use f.sat from L.Rigel, since this function ignores Alternatives. I therefore need a new function, called f.sat.alt.b. I define it later, as my aim now is simply to illustrate the ability to represent Alternatives in models.

Language: Mirfak

Language Modules L.Mirfak

r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, vr.alt.b, f.sat.inf.pos, f.sat.inf.neg, f.sat.alt.b, f.sat.leaf

Domain

There is a set of Fragments **F**, a singleton for Value Types

$$\mathbf{T} = \{\text{v.Satisfaction}\},$$

and a set of Value Assignments **V**. Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \text{c.r} \cup \text{c.k} \cup \text{c.s}$ and $\text{c.r} \cap \text{c.k} \cap \text{c.s} = \emptyset$. Influences are over Fragments, $\text{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\text{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

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

Satisfaction is binary, $\text{v.Satisfaction} = \{1,0\}$. Alternatives are over Value Assignments, $\text{vr.alt.b} \subseteq \mathbf{V} \times \mathbf{V}$.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$\begin{array}{lcl} A & ::= & x \mid y \mid z \mid \ldots \\ B & ::= & r(A) \mid k(A) \mid s(A) \\ C & ::= & B \text{ influences+ } B \\ D & ::= & B \text{ influences- } B \\ G & ::= & \langle A, E, F \rangle \\ H & ::= & G \text{ alternativeTo } G \\ Z & ::= & B \mid C \mid D \mid G \mid H \end{array}$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$. B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in \text{c.r}$, $\mathscr{D}(k(\alpha)) \in \text{c.k}$, $\mathscr{D}(s(\alpha)) \in \text{c.s}$. C and D symbols denote, respectively, positive and negative influence relations. E symbols denote Value Types, $\mathscr{D}(E) \in \mathbf{T}$. F symbols denote a value of a Value Type, and as there is one

Value Type, $\mathscr{D}(F) \in$ v.Satisfaction. G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$. H symbols denote Alternatives, $\mathscr{D}(H) \in$ vr.alt.b.

Language Services

Those of relations and functions in the language.

How should f.sat.alt.b work? Consider first why this function is needed. Suppose that you have the L.Mirfak model in Figure 12.1. Each node labeled "alt" is an instance of vr.alt.b. There are six such instances in the model.

Each vr.alt.b in the Figure represents a constraint on Outcomes of the model there. An Outcome which has, for example,

⟨AutoAmbList influences+ IdAmb, v.Satisfaction, 1⟩,
⟨ManTrckAmb influences+ IdAmb, v.Satisfaction, 1⟩

violates the vr.alt.b instance

⟨AutoAmbList influences+ IdAmb, v.Satisfaction, 1⟩ alternativeTo
⟨ManTrckAmb influences+ IdAmb, v.Satisfaction, 1⟩.

Incoherent Outcome

I will say that such an Outcome is Incoherent. More generally, if there is a vr.alt.b in a model, and an Outcome violates it, then that Outcome is Incoherent. An Outcome may be Incoherent for other reasons, which will be discussed later.

Figure 12.2 shows Value Assignments to leaf Fragments, made using f.sat.leaf. If you apply f.sat.inf.pos, the resulting Value Assignments will include

⟨AutoAmbList influences+ IdAmb, v.Satisfaction, 1⟩,
⟨UpdAutoAmbList influences+ IdAmb, v.Satisfaction, 1⟩,
⟨ManTrckAmb influences+ IdAmb, v.Satisfaction, 1⟩

and you can stop propagating v.Satisfaction values, as the resulting Outcome will inevitably be Incoherent. Figure 12.2 shows circles over the three problematic Value Assignments, whose propagation gives an Incoherent Outcome.

I will leave Incoherent Outcomes aside for now, and consider only Value Assignments which will give Outcomes that are Coherent

with regards to vr.alt.b. Figure 12.3 shows Value Assignments to leaf Fragments, which will give Coherent Outcomes. What should be the v.Satisfaction value of IdAmb and ChoAmb in Figure 12.3?

The Value Assignment in Figure 12.3 assures that IdAmb is satisfied. This is because AutoAmbList and UpdAutoAmbList are assumed satisfied via f.sat.leaf, and each positively influences IdAmb. Each of them is also an Alternative to ManTrckAmb, which is not satisfied. Any Outcome where AutoAmbList and UpdAutoAmbList are satisfied, and ManTrckAmb is not, will satisfy these two vr.alt.b instances. (There can be other Outcomes which can satisfy these two vr.alt.b instances. An Outcome which makes ManTrckAmb satisfied, but AutoAmbList and UpdAutoAmbList not satisfied, will also be Coherent with regards to the two vr.alt.b instances. I will be searching for all such Outcomes in Section 12.3.)

f.sat will not propagate appropriate v.Satisfaction values when there are vr.alt.b instances, as it would assign 0 to IdAmb in Figure 12.3. f.sat requires all incoming r.inf.pos instances to be satisfied, in order for their target Fragment to be satisfied.

f.sat.alt.b should not require all incoming positive influences to be satisfied, in order for the target to be satisfied. It should require a subset of incoming positive influences to be satisfied, as long as none of the members in this subset are themselves Alternatives. In other words, there should be no vr.alt.b instance over the members of such a subset. In the case of IdAmb, there are two candidate subsets, call them A_1 and A_2,

$$A_1 = \{\ \text{AutoAmbList influences+ IdAmb},\ \text{UpdAutoAmbList influences+ IdAmb}\ \},$$
$$A_2 = \{\ \text{ManTrckAmb influences+ IdAmb}\ \}.$$

If all members of *either* A_1 *or* A_2 are satisfied, then IdAmb should be satisfied as well. Notice my implicit assumption that *these sets should be the largest subsets* of positive influences. I will make this into a convention, since the idea with f.sat was that *all* positive influences should be satisfied, not some subset thereof. So now, I want to have as many non-Alternative positive influences satisfied and not, for example, at least one of them.

The conclusion of the above is that f.sat.alt.b should work as follows, when applied to compute the v.Satisfaction value of a Fragment x:

1. Find all positive and negative influence relation instances which target x. All these relation instances must have a satisfaction value assigned already. Let V_I be the set which includes v.Satisfaction Value Assignments to all these positive and negative relation instances.
2. Find all vr.alt.b instances over the Value Assignments in V_I. Let A_I be the set which includes all these vr.alt.b instances.
3. Find all the largest subsets of V_i, such that if o_i is such a subset, then there is no vr.alt.b instance over any pair of its members.
4. For each $o_i \in O$, compute the product of v.Satisfaction values in it.
5. If there is at least one $o_j \in O$, whose satisfaction value is 1, then $\langle x, \text{v.Satisfaction}, 1\rangle$, otherwise $\langle x, \text{v.Satisfaction}, 0\rangle$.

When applied to the model in Figure 12.3 and IdAmb, the first step gives

$$\begin{aligned} V_I = \{\ &\langle \text{AutoAmbList influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle, \\ &\langle \text{UpdAutoAmbList influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle, \\ &\langle \text{ManTrckAmb influences+ IdAmb}, \text{v.Satisfaction}, 0\rangle\ \}. \end{aligned}$$

Note that I had to propagate satisfaction values over the three influence relation instances first, in order to get the members of V_I. The second step gives

$$\begin{aligned} A_I = \{\ &\langle \text{AutoAmbList influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle \\ &\text{alternativeTo} \\ &\langle \text{ManTrckAmb influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle, \\ &\langle \text{UpdAutoAmbList influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle \\ &\text{alternativeTo} \\ &\langle \text{ManTrckAmb influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle\ \}. \end{aligned}$$

The third step results in

$$\begin{aligned} O &= \{\ o_1, o_2\ \}, \\ o_1 &= \{\ \langle \text{AutoAmbList influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle, \\ &\quad \langle \text{UpdAutoAmbList influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle\ \}, \\ o_2 &= \{\ \langle \text{ManTrckAmb influences+ IdAmb}, \text{v.Satisfaction}, 1\rangle\ \}. \end{aligned}$$

The fourth step calculates the product of satisfaction values in each Outcome, in O. The result is 1 for o_1 and 1 for o_2. Finally, the fifth step concludes with

$$\langle \mathsf{IdAmb}, \mathsf{v.Satisfaction}, 1 \rangle.$$

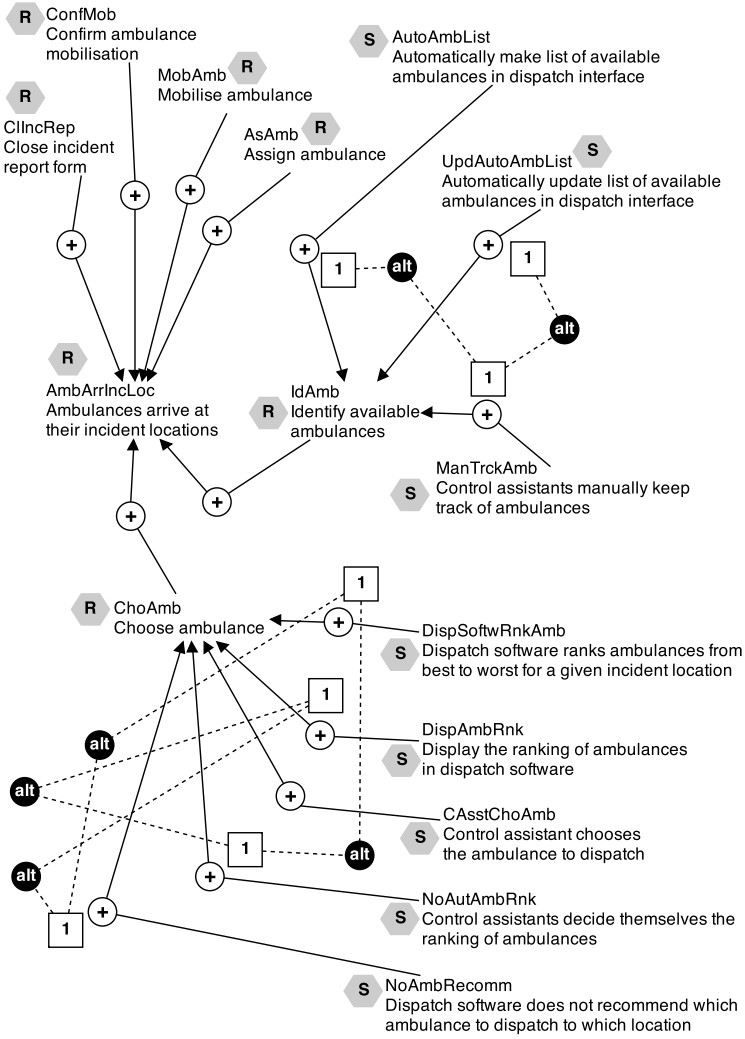


Figure 12.1: A visualisation of a model in L.Mirfak.

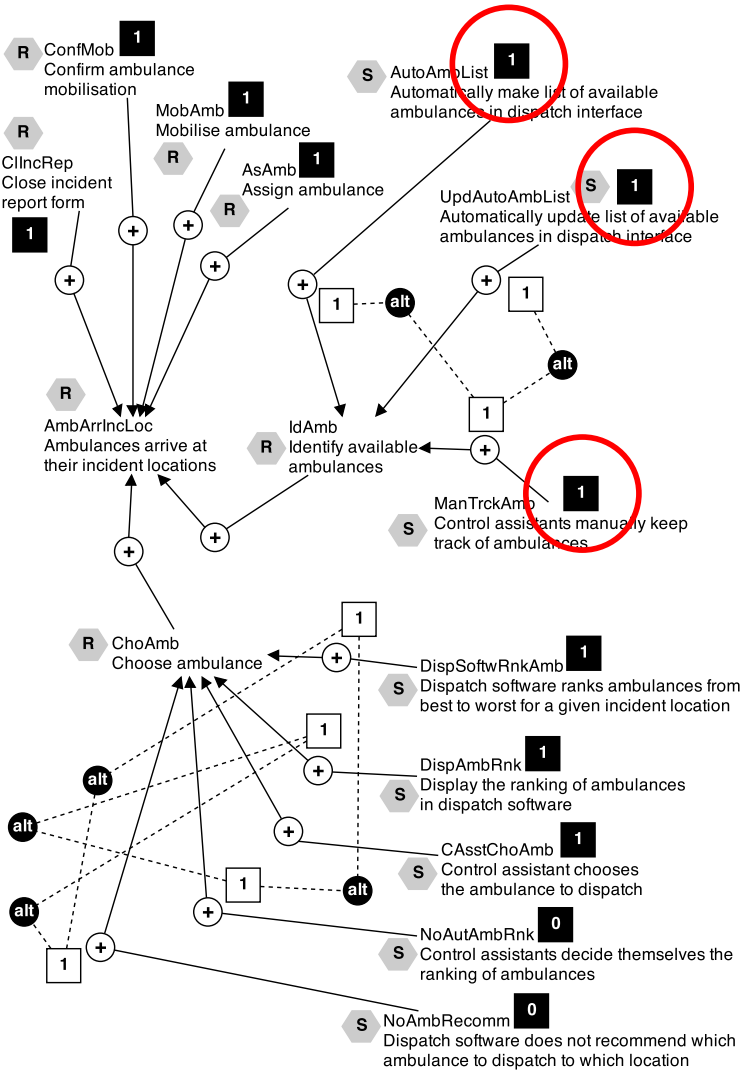


Figure 12.2: Value Assignments for an Incoherent Outcome.

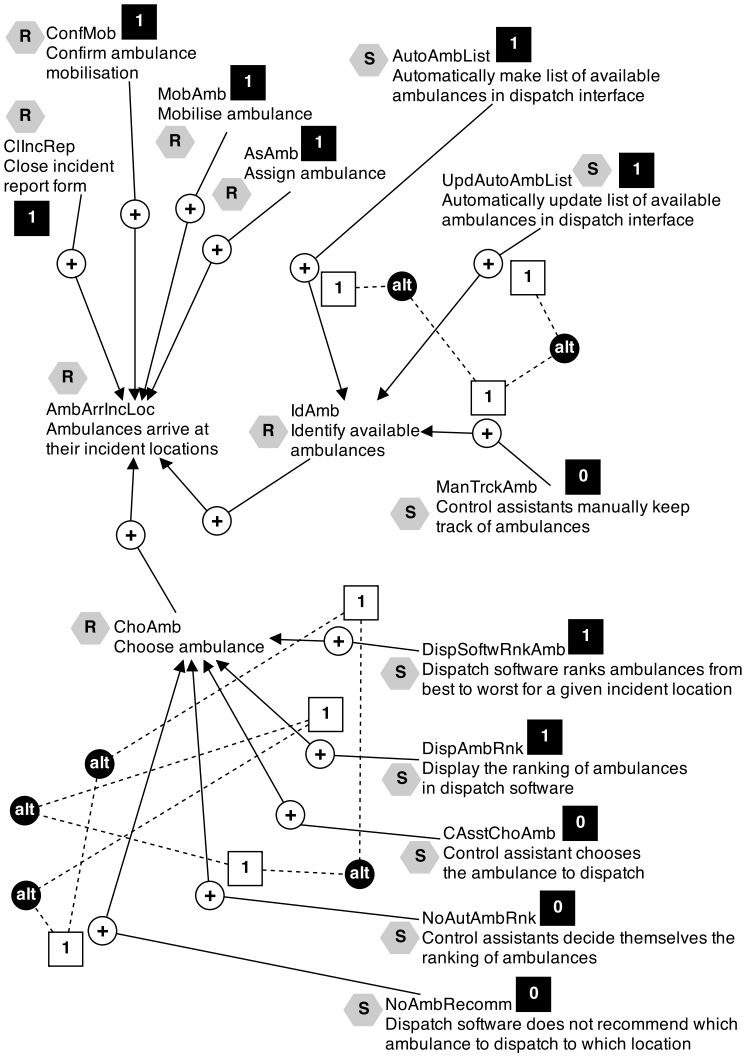


Figure 12.3: Value Assignments for an Incoherent Outcome.

The following is a definition of f.sat.alt.b, which works according to the rules above.

Function: sat.alt.b

Binary satisfaction in presence of Alternatives f.sat.alt.b

Input

Fragment or relation instance x, and model M.

Do

1. Find all positive and negative influence relation instances which target x. All these relation instances must have a v.Satisfaction value assigned already. Let V_I be the set which includes v.Satisfaction Value Assignments to all these positive and negative relation instances. That is, do the following:
   (a) Find the set $\{(p_1, x), \ldots, (p_n, x)\} \subseteq$ r.inf.pos of all positive influence relation instances to x in M. Call this set $I_+(x)$.
   (b) Find the set $\{(p_{n+1}, x), \ldots, (p_m, x)\} \subseteq$ r.inf.neg of all negative influence relation instances to x in M. Call this set $I_-(x)$.
   (c) For each $(p_i, x) \in I_+(x) \cup I_-(x)$, compute its satisfaction Value Assignment

   $$\langle (p_i, x), \text{v.Satisfaction}, 1 \rangle = \text{f.sat.alt.b}((p_i, x), M),$$

   and add this Value Assignment to V_I.
2. Find all vr.alt.b instances over the Value Assignments in V_I. Let A_I be the set which includes all these vr.alt.b instances. Thus, A_I includes all vr.alt.b instances with either this

format

$$\langle p_i \text{ influences+ } x, \text{v.Satisfaction}, 1\rangle \text{ alternativeTo } \langle p_j \text{ influences+ } x, \text{v.Satisfaction}, 1\rangle$$

or this format

$$\langle p_i \text{ influences- } x, \text{v.Satisfaction}, 1\rangle \text{ alternativeTo } \langle p_j \text{ influences- } x, \text{v.Satisfaction}, 1\rangle$$

where both (p_i, x) and (p_j, x) are members of $I_+(x) \cup I_-(x)$.

3. Find all the largest subsets of V_i, such that if o_i is such a subset, then there is no vr.alt.b instance over any pair of its members.
4. For each $o_i \in O$, compute the product of v.Satisfaction values in it. Let $v(o_i)$ be that value.
5. If there is at least one $o_j \in O$, whose satisfaction value is 1, then let $s = 1$, otherwise let $s = 0$.

Output

$\langle x, \text{v.Satisfaction}, s\rangle$.

Language Services

- s.WhSat: $\langle x, \text{v.Satisfaction}, s\rangle$.

Figure 12.4 shows a visualisation of a model in L.Mirfak. The Value Assignments there show one Coherent Outcome, made by applying f.sat.leaf, f.sat.inf.pos, f.sat.if.neg, and f.sat.alt.b.

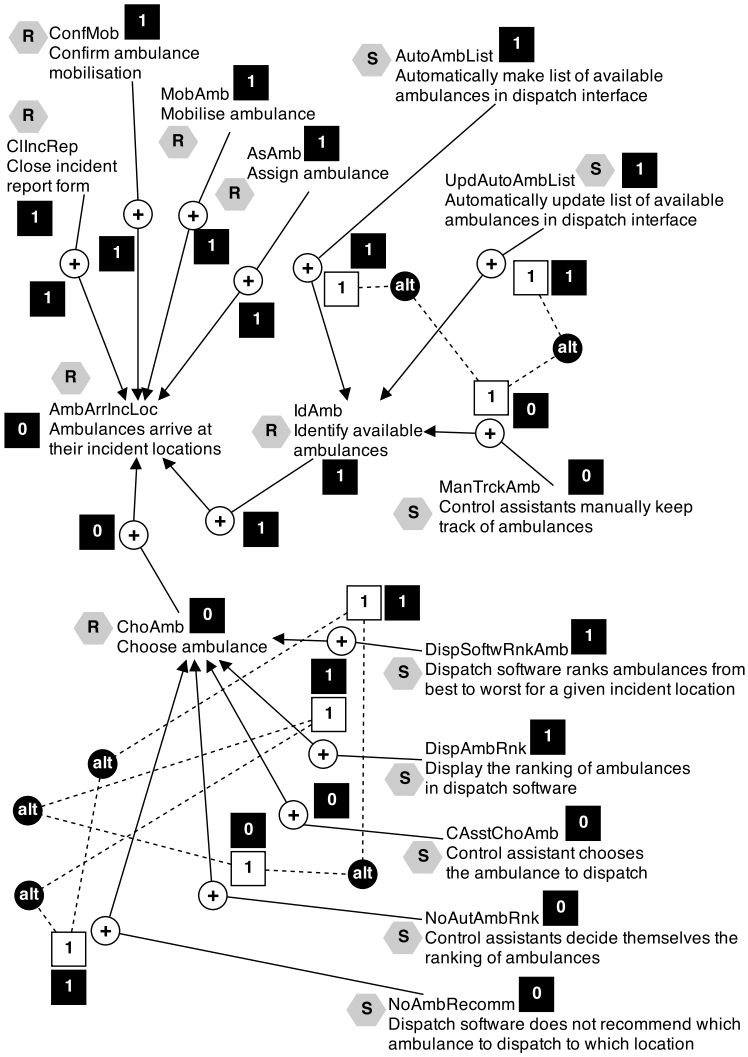


Figure 12.4: A Coherent Outcome in a L.Mirfak model.

12.3 Picks and their Use

A Pick, denoted P, is an Outcome which satisfies the following conditions: Pick

1. It is a Coherent Outcome,
2. It includes only Value Assignments which are for some reason desirable to you.

The idea is that you first define a Pick for a model, and then search for Outcomes which include that Pick. It may be that a model can have Outcomes which include the Pick, but it can also happen that there are no such Outcomes. If, for example, the Pick includes all requirements in a model, and the model itself shows various ways of satisfying these requirements, then looking for Outcomes which include this Pick amounts to looking for Outcomes which ensure that the requirements are satisfied.

There are no constraints on what goes in P. It can, for example, include Value Assignments over different Value Types. There are therefore different Picks, depending on the content of P.

To illustrate how to use Picks, and what for, consider how the following Language Service can be delivered.

Language Service: MandSat

Given a model in which every Fragment can be assigned a binary satisfaction value, and a binary importance value, which are all the Complete and Coherent Outcomes of that model, in which all mandatory Fragments are satisfied? s.MandSat

A Complete Outcome has a Value Assignment to every variable in a model. Recall that there is one variable per pair of Fragment and Value Type. Complete Outcome

s.MandSat can work with a language with two Value Types. One is a binary satisfaction value, and v.Satisfaction will do. The other is binary importance value, which I will call v.Importance. Importance value is either 1, if satisfying the Fragment or relation instance is mandatory, or 0 otherwise. It follows that s.MandSat consists of

finding all Outcomes of a model, which are a superset of the following Pick:

$$\mathsf{P} = \{\, \langle x, \text{v.Satisfaction}, 1\rangle \mid \forall\, x \in M \text{ s.t.} \\ x \in \mathbf{F} \text{ and } \langle x, \text{v.Importance}, 1\rangle \,\}.$$

Having defined the Pick, I need a language whose Outcomes can have it as a subset. Let L.Pollux be the language made by adding v.Importance via f.imp.asm to L.Mirfak. For simplicity, there is no propagation of importance values. They are assigned manually to individual Fragments only, not to relation instances using f.imp.asm. The new function is defined as follows.

Function: imp.asm	
Assume an importance value for a Fragment	f.imp.asm
Input Fragment x and model M.	
Do If you assume that x must be satisfied, then $v = 1$, else $v = 0$.	
Output $\langle x, \text{v.Importance}, v\rangle$	
Language Services • **s.WhImpAsm**: Which, if any, is the assumed v.Importance value of x in M? : $\langle x, \text{v.Importance}, v\rangle$.	s.WhImpAsm

The language is made by adding f.imp.asm to L.Mirfak. Syntax and mapping remain the same, the domain changes, as it now has v.Importance.

Language: Pollux

Language Modules L.Pollux

r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, vr.alt.b, f.sat.inf.pos, f.sat.inf.neg, f.sat.alt.b, f.sat.leaf, f.imp.asm

Domain

There is a set of Fragments **F** and Value Types

$$\mathbf{T} = \{\text{v.Satisfaction}, \text{v.Importance}\},$$

and a set of Value Assignments **V**. Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \text{c.r} \cup \text{c.k} \cup \text{c.s}$ and $\text{c.r} \cap \text{c.k} \cap \text{c.s} = \emptyset$. Influences are over Fragments, $\text{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\text{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$. Satisfaction value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

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

Importance Value Assignments are over Fragments only, so that

$$\mathbf{V} \subseteq \mathbf{F} \times \{\text{v.Importance}\} \times \text{v.Importance}.$$

Both the satisfaction and importance Value Types are binary, $\text{v.Satisfaction} = \text{v.Importance} = \{1, 0\}$. Alternatives are over Value Assignments of the same Value Type.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$A ::= x \mid y \mid z \mid \ldots$$
$$B ::= r(A) \mid k(A) \mid s(A)$$
$$C ::= B \text{ influences+ } B$$
$$D ::= B \text{ influences- } B$$

$$G ::= \langle A, E, F \rangle$$
$$H ::= G \text{ alternativeTo } G$$
$$Z ::= B \mid C \mid D \mid G \mid H$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in$ **F**. B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. C and D symbols denote, respectively, positive and negative influence relations. E symbols denote Value Types, $\mathscr{D}(E) \in$ **T**. F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction. G symbols denote Value Assignments, $\mathscr{D}(G) \in$ **V**. H symbols denote Alternatives, $\mathscr{D}(H) \in$ vr.alt.b.

Language Services

Those of relations and functions in the language.

Figure 12.5 shows a visualisation of a model in L.Pollux. As there are two Value Types, labels are different than in the visualisation of L.Mirfak models. Now, label "s1" is for the assignment of the satisfaction value 1, and "s0" if the satisfaction value is 0. "i1" is the assignment of the importance value 1, and "i0" of value 0. According to the model in the Figure, ChoAmb is the only Fragment which must be satisfied. Therefore, the Pick is

$$\mathsf{P} = \{ \langle \text{AmbArrIncLoc}, \text{v.Satisfaction}, 1 \rangle, \langle \text{AmbArrIncLoc}, \text{v.Importance}, 1 \rangle \}$$

that is, the Outcomes to find should assign the satisfaction value 1 and importance value 1 to AmbArrIncLoc.

How, then, to find all Outcomes in of the model in Figure 12.5, which include the Pick P above? In other words, define a function which takes a model in L.Pollux and a Pick, and returns all Outcomes supersets of that Pick, if they exist, or an empty set if there are none.

This function will have to assign values in a different way propagation. It should also not produce Outcomes which contradict

those that would have been produced on a model, if satisfaction values were propagated from the leaves, using f.sat.inf.pos, f.sat.inf.neg, f.sat.leaf, and f.sat, since these functions are part of L.Pollux.

In the model in Figure 12.5, AmbArrIncLoc must be satisfied, and therefore must have the satisfaction value 1. It will have that value only if all positive influences to it, and all negative influences to it have the satisfaction value 1. Otherwise, I would be violating the rules of f.sat.inf.pos, f.sat.inf.neg, and f.sat. It follows that any Outcome which includes P must also include the following Value Assignments:

$$\{\langle \text{CIIncRep influences+ AmbArrIncLoc}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{ConfMob influences+ AmbArrIncLoc}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{MobAmb influences+ AmbArrIncLoc}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{AsAmb influences+ AmbArrIncLoc}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{IdAmb influences+ AmbArrIncLoc}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{ChoAmb influences+ AmbArrIncLoc}, \text{v.Satisfaction}, 1\rangle \}.$$

According to f.sat.inf.pos, the an instance of a positive influence relation will have the satisfaction vale 1 only if its origin Fragment also has the satisfaction value 1. It follows that all Outcomes which include P must also include these Value Assignments:

$$\{\langle \text{CIIncRep}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{ConfMob}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{MobAmb}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{AsAmb}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{IdAmb}, \text{v.Satisfaction}, 1\rangle,$$
$$\langle \text{ChoAmb}, \text{v.Satisfaction}, 1\rangle \}.$$

The resulting Outcome is shown in Figure 12.6. It is an Incomplete Outcome.

There are Alternatives for satisfying IdAmb in Figure 12.6. You can find them by solving a system of equations over variables defined by the positive influence relations to IdAmb, and the Fragments in which these relation instances originate. To simplify notation and write w_1 for the variable

$$\text{AutoAmbList.v.Satisfaction},$$

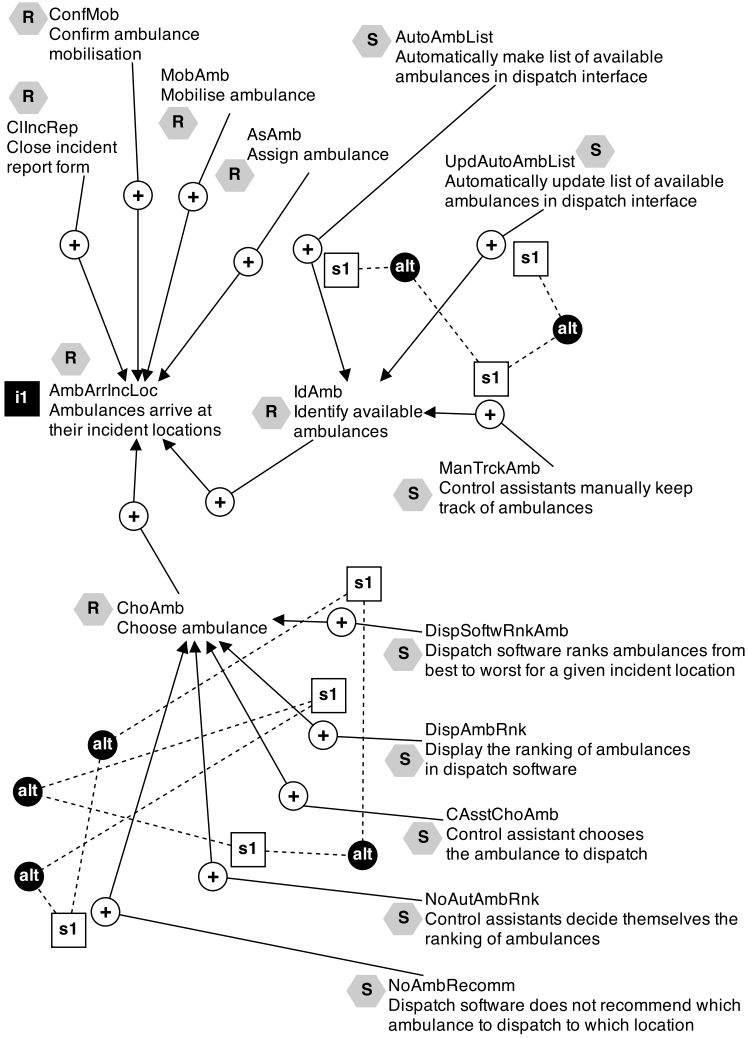


Figure 12.5: A L.Pollux model.

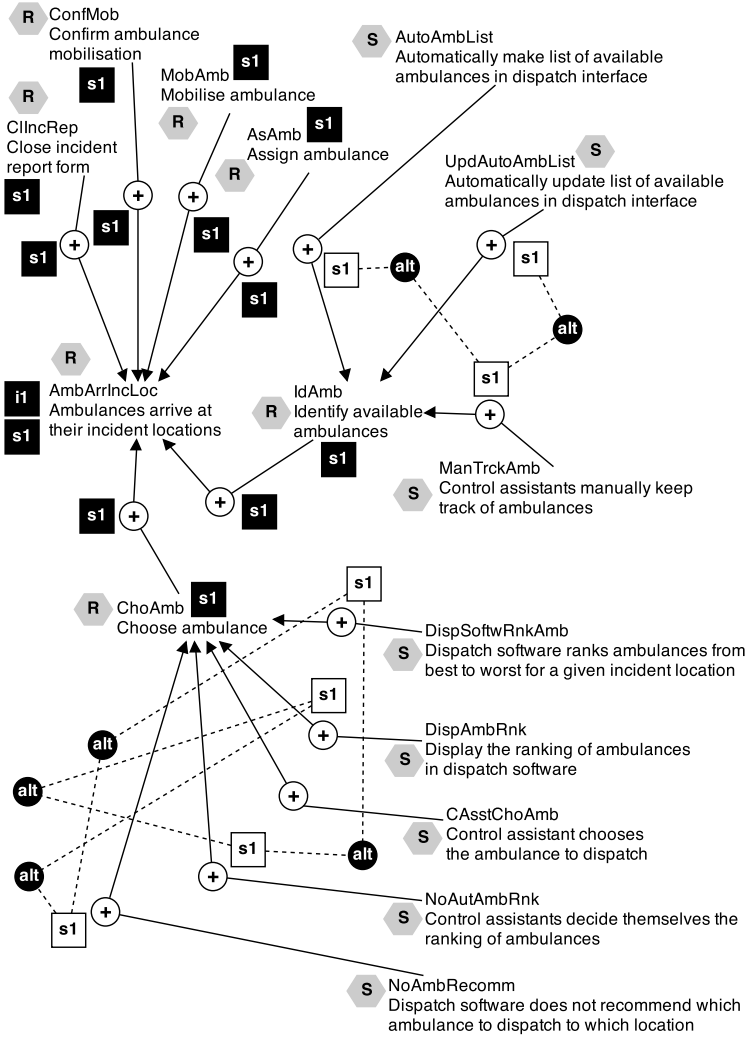


Figure 12.6: A L.Pollux model.

use the following equivalences

$$
\begin{aligned}
w_1 &\equiv \text{AutoAmbList.v.Satisfaction},\\
w_{1.4} &\equiv \text{(AutoAmbList influences+ IdAmb).v.Satisfaction},\\
w_2 &\equiv \text{UpdAutoAmbList.v.Satisfaction},\\
w_{2.4} &\equiv \text{(UpdAutoAmbList influences+ IdAmb).v.Satisfaction},\\
w_3 &\equiv \text{ManTrckAmb.v.Satisfaction},\\
w_{3.4} &\equiv \text{(ManTrckAmb influences+ IdAmb).v.Satisfaction},\\
w_7 &\equiv \text{IdAmb.v.Satisfaction}.
\end{aligned}
$$

The system of equations is then as follows:

$$
\begin{aligned}
0 &= w_1 - w_{1.4},\\
0 &= w_2 - w_{2.4},\\
0 &= w_3 - w_{3.4},\\
1 &= |w_{1.4} - w_{3.4}|,\\
1 &= |w_{2.4} - w_{3.4}|,\\
w_4 &= |w_{1.4} * w_{2.4} - w_{3.4}|.
\end{aligned}
$$

The first three equations above reflect the rules in f.sat.inf.pos. If $w_1 = 1$, that, is if AutoAmbList is satisfied, then the positive influence from it, to IdAmb, must be satisfied as well, that is, $w_{1.4} = 1$. If $w_1 = 0$, then $w_{1.4} = 0$, and *vice versa*. The fourth and fifth equations are due to r.alt.b instances. If $w_1 = w_{1.4} = 1$, then $w_3 * w_{3.4}$ has to be 0 according to the fourth equation. The sixth equation is due to the rule in f.sat.alt.b, which requires that the largest subset of non-Alternative influence relation instances be satisfied, in order for their target to be satisfied.

If you set $w_7 = 1$, there are two solutions to the system of equations above. They are

$$
\begin{aligned}
&w_1 = 1, w_{1.4} = 1, w_2 = 1, w_{2.4} = 1, w_3 = 0, w_{3.4} = 0, \text{ and}\\
&w_1 = 0, w_{1.4} = 0, w_2 = 0, w_{2.4} = 0, w_3 = 1, w_{3.4} = 1.
\end{aligned}
$$

The system of equations above is specific to the paths which end in IdAmb in Figure 12.6. There is another system of equations for Alternatives to ChoAmb. Again, for simplicity, start with these equiv-

alences:

$u_1 \equiv$ DispSoftwRnkAmb.v.Satisfaction,
$u_2 \equiv$ DispAmbRnk.v.Satisfaction,
$u_3 \equiv$ CAsstChoAmb.v.Satisfaction,
$u_4 \equiv$ NoAutAmbRnk.v.Satisfaction,
$u_5 \equiv$ NoAmbRecomm.v.Satisfaction,
$u_6 \equiv$ ChoAmb.v.Satisfaction,
$u_{1.6} \equiv$ (DispSoftwRnkAmb influences+ ChoAmb).v.Satisfaction,
$u_{2.6} \equiv$ (DispAmbRnk influences+ ChoAmb).v.Satisfaction,
$u_{3.6} \equiv$ (CAsstChoAmb influences+ ChoAmb).v.Satisfaction,
$u_{4.6} \equiv$ (NoAutAmbRnk influences+ ChoAmb).v.Satisfaction,
$u_{5.6} \equiv$ (NoAmbRecomm influences+ ChoAmb).v.Satisfaction.

ChoAmb has to be satisfied, that is, $u_6 = 1$, in order for AmbArrIncLoc to be satisfied. You can consequently find all Outcomes which include the Fragments and positive influences above by solving the following system of equations:

$$\begin{aligned}
0 &= u_1 - u_{1.6},\\
0 &= u_2 - u_{2.6},\\
0 &= u_3 - u_{3.6},\\
0 &= u_4 - u_{4.6},\\
0 &= u_5 - u_{5.6},\\
1 &= |u_{1.6} - u_{5.6}|,\\
1 &= |u_{1.6} - u_{4.6}|,\\
1 &= |u_{2.6} - u_{5.6}|,\\
1 &= |u_{2.6} - u_{4.6}|,\\
u_6 &= u_{3.6} * |(u_{1.6} * u_{2.6} - u_{4.6} * u_{5.6})|.
\end{aligned}$$

There are two solutions. One is

$$\begin{aligned}
u_1 = u_{1.6} = u_2 = u_{2.6} = u_3 = u_{3.6} = 1,\\
u_4 = u_{4.6} = u_5 = u_{5.6} = 0,
\end{aligned}$$

and the other is

$$\begin{aligned}
u_1 = u_{1.6} = u_2 = u_{2.6} = 0,\\
u_3 = u_{3.6} = u_4 = u_{4.6} = u_5 = u_{5.6} = 1.
\end{aligned}$$

As there are two solutions for each system of equations, and there are two such systems in the model in Figure 12.5, it follows that there are four Complete Outcomes of that model, each of which is Coherent with regards to f.sat.inf.pos, f.sat.inf.neg, f.sat.leaf, and f.sat.alt.b, and is a superset of P. The four are shown in Figures 12.7–12.10. In each Figure, the circles highlight the Fragments and relation instances which are not satisfied.

The following pages show the Figures mentioned above. After the Figures, I discuss a function which can produce these systems of equations for L.Pollux models.

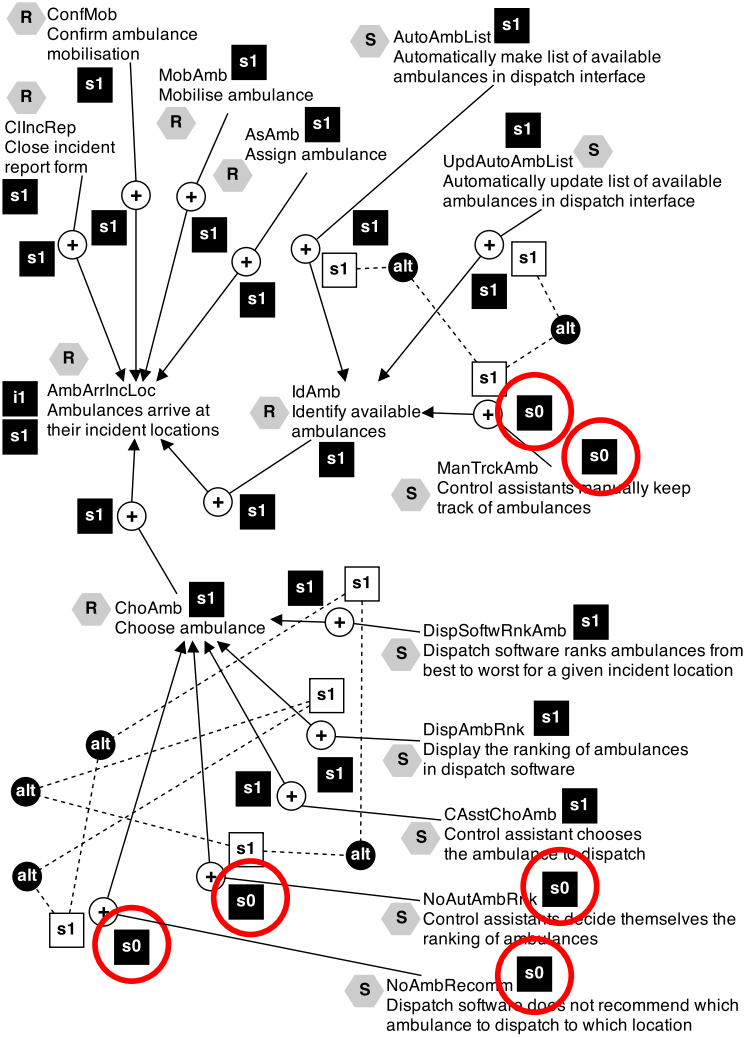


Figure 12.7: One Complete Outcome.

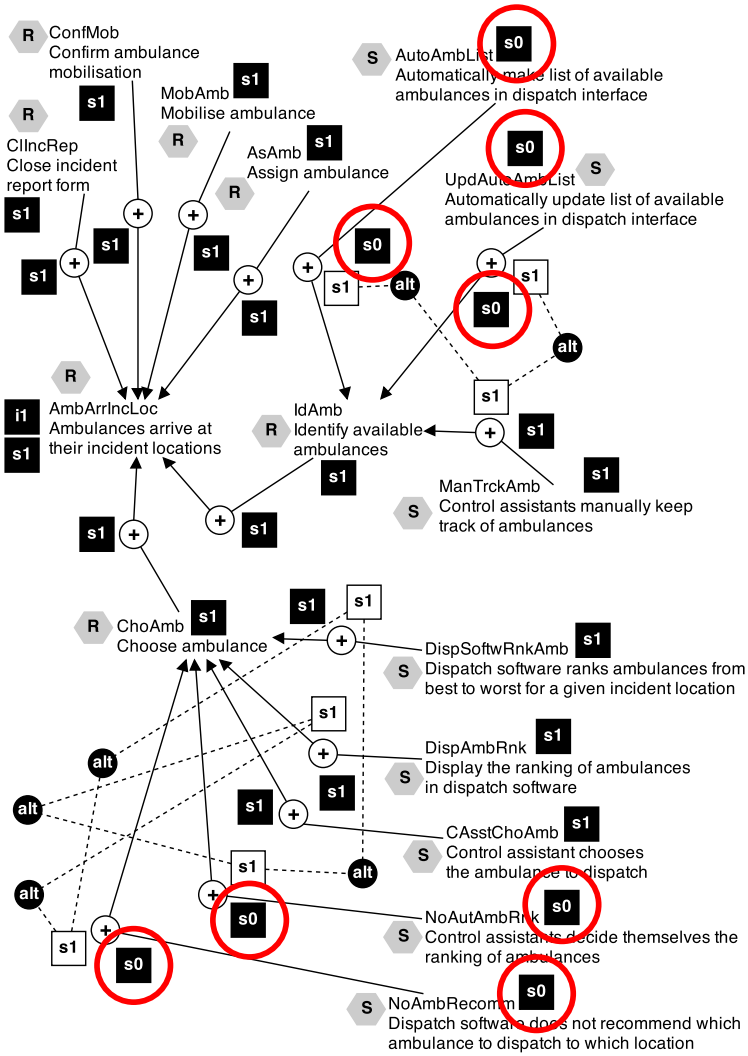


Figure 12.8: Second Complete Outcome.

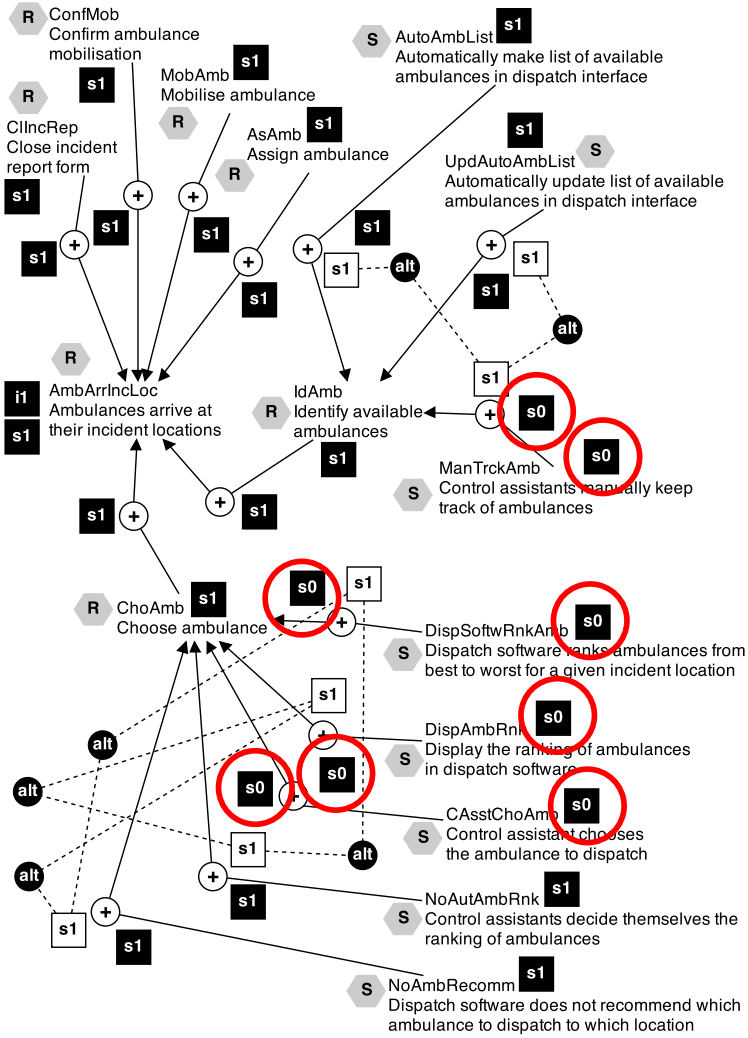


Figure 12.9: Third Complete Outcome.

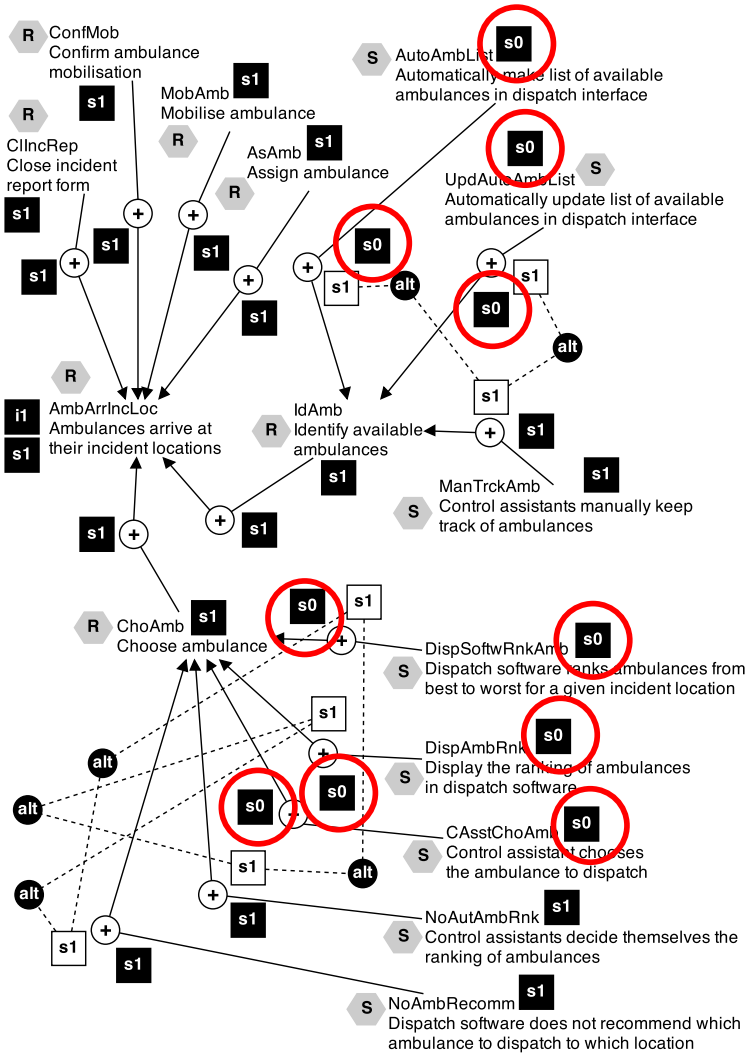


Figure 12.10: Fourth Complete Outcome.

To find all Outcomes which include a particular Pick, and do this for models of L.Pollux, I need a function which produces the systems of equations, in the same way I did for the model in Figure 12.5, then solves them, and finally, returns the Outcomes. Here is a sketch of how the function could work, based on what I did for the model in Figure 12.5:

1. Take a L.Pollux model M, and make a directed hypergraph $H(M)$ from it, such that in $H(M)$ there is one node for every Fragment in M, and a directed edge for every positive and every negative influence relation instance, in the direction of influence.

   Example 12.3.1. Figure 12.5 shows the graph for a model in L.Pollux. It has at most one edge between any two nodes, and is therefore not a hypergraph. •

2. Check if $H(M)$ includes cycles:

   - If yes, then stop, because the model M is Incoherent with regards to r.inf.pos and, or r.inf.neg. These two relations are irreflexive and transitive, and therefore, cycles are not allowed in the hypergraph which these relations induce over Fragments.
   - If there are no cycles, then go to next step.

   Example 12.3.2. *(Example 12.3.1 continued.)* The graph in Figure 12.5 has no cycles. Note that it is a tree, as it is acyclic and has one root. •

3. For each Fragment x in M, define a variable w_x

   $$w_x \equiv x.\text{v.Satisfaction}$$

   which takes a satisfaction value, and add that variable to the set of all variables from M, denoted W_M.

   Example 12.3.3. *(Example 12.3.2 continued.)* W_M includes one variable per Fragment in Figure 12.5:

   $$\begin{aligned} W_M = \{\ & w_{\text{AmbArrIncLoc}}, w_{\text{CIIncRep}}, w_{\text{ConfMob}} \\ & w_{\text{MobAmb}}, w_{\text{AsAmb}}, w_{\text{IdAmb}}, \\ & w_{\text{AutoAmbList}}, w_{\text{UpdAutoAmbList}}, w_{\text{ManTrckAmb}}, \\ & w_{\text{ChoAmb}}, w_{\text{DispSoftwRnkmb}}, w_{\text{DispAmbRnk}}, \\ & w_{\text{AsstChoAmb}}, w_{\text{NoAutAmbRnk}}, w_{\text{NoAmbRecomm}}\ \}. \end{aligned}$$

•

4. For each r.inf.pos instance x influences+ y, from Fragment x to Fragment y,

   (a) define a variable $w_{x.p.y}$

   $$w_{x.p.y} \equiv (x \text{ influences+ } y).\text{v.Satisfaction},$$

   and add this variable to W_M,

   (b) define an equation

   $$w_x - w_{x.p.y} = 0$$

   which requires that w_x and $w_{x.p.y}$ have the same v.Satisfaction value, following the rules in f.sat.inf.pos, and add this equation to the set of equations E_M.

Example 12.3.4. *(Example 12.3.3 continued.)* To simplify notation, I introduce the following equivalences:

$$\begin{aligned}
w_1 &\equiv w_{\text{AmbArrIncLoc}}, & w_2 &\equiv w_{\text{CllncRep}},\\
w_3 &\equiv w_{\text{ConfMob}}, & w_4 &\equiv w_{\text{MobAmb}},\\
w_5 &\equiv w_{\text{AsAmb}}, & w_6 &\equiv w_{\text{IdAmb}},\\
w_7 &\equiv w_{\text{AutoAmbList}}, & w_8 &\equiv w_{\text{UpdAutoAmbList}},\\
w_9 &\equiv w_{\text{ManTrckAmb}}, & w_{10} &\equiv w_{\text{ChoAmb}},\\
w_{11} &\equiv w_{\text{DispSoftwRnkmb}}, & w_{12} &\equiv w_{\text{DispAmbRnk}},\\
w_{13} &\equiv w_{\text{AsstChoAmb}}, & w_{14} &\equiv w_{\text{NoAutAmbRnk}},\\
w_{15} &\equiv w_{\text{NoAmbRecomm}}.
\end{aligned}$$

Using these equivalences, this first part of this fourth step gives the following new variables for W_M, one per positive influence relation instance in Figure 12.5

$$\begin{aligned}
&w_{2.p.1}, w_{3.p.1}, w_{4.p.1}, w_{5.p.1}, w_{6.p.1}, w_{10.p.1},\\
&w_{7.p.6}, w_{8.p.6}, w_{9.p.6},\\
&w_{11.p.10}, w_{12.p.10}, w_{13.p.10}, w_{14.p.10}, w_{15.p.10}.
\end{aligned}$$

The second part of this fourth step gives the following equations for E_M

$$
\begin{aligned}
0 &= w_2 - w_{2.p.1}, & 0 &= w_3 - w_{3.p.1},\\
0 &= w_4 - w_{4.p.1}, & 0 &= w_5 - w_{5.p.1},\\
0 &= w_6 - w_{6.p.1}, & 0 &= w_{10} - w_{10.p.1},\\
0 &= w_7 - w_{7.p.6}, & 0 &= w_8 - w_{8.p.6},\\
0 &= w_9 - w_{9.p.6}, & & \\
0 &= w_{11} - w_{11.p.10}, & 0 &= w_{12} - w_{12.p.10},\\
0 &= w_{13} - w_{13.p.10}, & 0 &= w_{14} - w_{14.p.10},\\
0 &= w_{15} - w_{15.p.10}. & &
\end{aligned}
$$

At this point, all Fragments and positive influence relation instances have corresponding variables which take a satisfaction value. •

5. For each r.inf.neg instance x influences- y, from Fragment x to Fragment y,

   (a) define a variable $w_{x.n.y}$

   $$w_{x.n.y} \equiv (x \text{ influences- } y).\text{v.Satisfaction},$$

   and add this variable to W_M,

   (b) define an equation

   $$|w_x - w_{x.n.y}| = 1$$

   which follows the rules f.inf.neg, and requires that if $w_x = 1$ then $w_{x.n.y} = 0$, and *vice versa*. Add this equation to the set of equations E_M.

At this point, you have the set W_M which includes a variable for every Fragment and every positive and negative influence relation instance in M, and the set E_M of equations, one per positive and negative influence relation instance. You still need equations which correspond to r.alt.b instances.

Example 12.3.5. *(Example 12.3.4 continued.)* There are no negative influence relation instances in Figure 12.5. This step therefore does not change W_M and E_M updated in the fifth step. •

6. For each r.alt.b instance over values of two variables $w_i \in W_M$ and $w_j \in W_M$,

   - if the instance is

     $$(w_i = 1) \text{ alternativeTo } (w_j = 1),$$

     then add an equation

     $$|w_i - w_j| = 1$$

     to the set E_M,
   - if the instance is

     $$(w_i = 1) \text{ alternativeTo } (w_j = 0),$$

     then add an equation

     $$w_i - w_j \leq 0$$

     to the set E_M,
   - if the instance is

     $$(w_i = 0) \text{ alternativeTo } (w_j = 1),$$

     then add an equation

     $$w_i - w_j \geq 0$$

     to the set E_M,
   - if the instance is

     $$(w_i = 0) \text{ alternativeTo } (w_j = 0),$$

     then add an equation

     $$|w_i - w_j| \geq 0$$

     to the set E_M.

Example 12.3.6. *(Example 12.3.5 continued.)* All r.alt.b instances are between the assignments of satisfaction value 1,

so that only $|w_i - w_j| = 1$ equations need to be added to E_M. These equations are

$$1 = |w_{7.p.6} - w_{9.p.6}|, \qquad 1 = |w_{8.p.6} - w_{9.p.6}|,$$
$$1 = |w_{11.p.10} - w_{14.p.10}|, \qquad 1 = |w_{11.p.10} - w_{15.p.10}|,$$
$$1 = |w_{12.p.10} - w_{14.p.10}|, \qquad 1 = |w_{12.p.10} - w_{15.p.10}|.$$

•

7. For each Fragment x in M, do the following:

   (a) Make the set W_M^x, so that it only includes all variables of the format $w_{y.p.x}$ and $w_{y.n.x}$, where y is any Fragment in M, so that W_M^x includes all variables for positive and negative influence relation instances which end in x.

   Example 12.3.7. *(Example 12.3.6 continued.)* Of all Fragments in Figure 12.5, only three are targets of influence relation instances, so there are only three sets W_M^x, as follows:

$$W_M^{\mathsf{AmbArrIncLoc}} = \{\, w_{2.p.1}, w_{3.p.1}, w_{4.p.1}, w_{5.p.1}, w_{6.p.1}, w_{10.p.1} \,\},$$
$$W_M^{\mathsf{IdAmb}} = \{\, w_{7.p.6}, w_{8.p.6}, w_{9.p.6} \,\},$$
$$W_M^{\mathsf{ChoAmb}} = \{\, w_{11.p.10}, w_{12.p.10}, w_{13.p.10}, w_{14.p.10}, w_{15.p.10} \,\}.$$

   •

   (b) Make the set O_M^x, which includes all the largest subsets of W_M^x which include no Alternatives, that is, every member o_i^x of O_M^x is the largest subset of W_M^x in which there are no two variables, which represent mutually exclusive influence relation instances.

Example 12.3.8. *(Example 12.3.7 continued.)* The sets are

$$\begin{aligned}
O_M^{\mathsf{AmbArrIncLoc}} &= \{\, o_1^{\mathsf{AmbArrIncLoc}} \,\},\\
o_1^{\mathsf{AmbArrIncLoc}} &= W_M^{\mathsf{AmbArrIncLoc}},\\
O_M^{\mathsf{IdAmb}} &= \{\, o_1^{\mathsf{IdAmb}}, o_2^{\mathsf{IdAmb}} \,\},\\
o_1^{\mathsf{IdAmb}} &= \{\, w_{7.p.6}, w_{8.p.6} \,\},\\
o_2^{\mathsf{IdAmb}} &= \{\, w_{9.p.6} \,\},\\
O_M^{\mathsf{ChoAmb}} &= \{\, o_1^{\mathsf{ChoAmb}}, o_2^{\mathsf{ChoAmb}} \,\},\\
o_1^{\mathsf{ChoAmb}} &= \{\, w_{11.p.10}, w_{12.p.10}, w_{13.p.10} \,\},\\
o_2^{\mathsf{ChoAmb}} &= \{\, w_{13.p.10}, w_{14.p.10}, w_{15.p.10} \,\}.
\end{aligned}$$

•

(c) Add a formula

$$\sum_{o_i^x \in O_M^x} \left(\prod_{w_{y.p.x} \in o_i^x} w_{y.p.x} \cdot \prod_{w_{y.n.x} \in o_i^x} w_{y.n.x} \right) = w_x$$

to E_M, which is used to indicate that only one of all the largest non-Alternative sets of influence relation instances to x should be satisfied in one Outcome.

Example 12.3.9. *(Example 12.3.8 continued.)* The three equations to add to E_M are

$$\begin{aligned}
w_{2.p.1} \cdot w_{3.p.1} \cdot w_{4.p.1} \cdot w_{5.p.1} \cdot w_{6.p.1} \cdot w_{10.p.1} &= w_1,\\
w_{7.p.6} \cdot w_{8.p.6} + w_{9.p.6} &= w_6,\\
w_{11.p.10} \cdot w_{12.p.10} \cdot w_{13.p.10} & \\
+ w_{13.p.10} \cdot w_{14.p.10} \cdot w_{15.p.10} &= w_{10}.
\end{aligned}$$

•

8. Take the Pick you want to find Outcomes for, and add each Value Assignment $w_i = v$ in the Pick to the set of equations E_M.

   Example 12.3.10. *(Example 12.3.9 continued.)* Suppose that the Pick is

   $$\mathsf{P} = \{\, \langle \mathsf{AmbArrIncLoc}, \mathsf{v.Satisfaction}, 1 \rangle \,\},$$

   and consequently, add $w_1 = 1$ to E_M. •

9. Find all solutions to the system of equations defined by E_M. Each solution is a Complete Outcome of M.

The above remains a sketch of a function. Its aim is to illustrate how you could generate systems of equations to solve, to find all Outcomes which include a Pick of interest. I do not provide a proof that it works correctly with any L.Pollux model, and I do not discuss how hard it is to actually solve systems of equations it gives, for models of any size. That is, I do not discuss reasoning complexity. These issues are important when defining a new language. I leave them outside this book, as they require deeper expertise in mathematics and formal logic than I can offer.

12.4 Several Arbitrary Value Types

This section focuses on how Alternatives and Picks can be used when there several arbitrary Value Types in a language. Consider this Language Service:

Language Service: AccTime

According to the model M, what is the estimated time required to make a system that satisfies those mandatory requirements, for which the specifications are in the product roadmap?

s.AccTime

s.AccTime can be relevant for a team which needs to

- document requirements, domain knowledge, and specifications,
- distinguish mandatory requirements from others,
- keep track of the progress in implementing specifications, and
- have an estimate of time to implement specifications, which are mature enough to be in the roadmap, and are necessary to satisfy the mandatory requirements.

The language for this team needs to have f.WhichKSR, so as to distinguish requirements, domain knowledge, and specifications in models. v.ProgrStatus from Section 10.5 can be used to keep track of progress in implementing specifications. Statuses can be assigned in the sequence prescribed in f.chk.progrstatus. The language also needs a Value Type for satisfaction, in order to identify the satisfied mandatory requirements. v.Satisfaction will do. It needs a Value Type to distinguish mandatory from other requirements, and it can use v.Importance. Finally, it should be possible to assign time estimates to specifications, and v.ImplTime can be used.

The resulting language has several arbitrary Value Types, in the sense that v.Satisfaction and v.Importance are binary, v.ProgrStatus has values on a nominal scale and comes with constraints defined f.chk.progrstatus, and v.ImplTime takes a positive real value. The language is called L.Avior and is defined below.

Language: Avior

Language Modules L.Avior

r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, vr.alt.b, f.sat.inf.pos, f.sat.inf.neg, f.sat.alt.b, f.sat.leaf, f.imp.asm, f.chk.progrstatus

Domain

- The domain is made of a set of Fragments **F**, relation instances **R**, Value Types **T**, and Value Assignments **V**.
- Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \text{c.r} \cup \text{c.k} \cup \text{c.s}$ and $\text{c.r} \cap \text{c.k} \cap \text{c.s} = \emptyset$.
- Relation instances are binary and either over Fragments or Value Assignments, $\mathbf{R} = (\mathbf{F} \times \mathbf{F}) \cup (\mathbf{V} \times \mathbf{V})$, and are partitioned as follows:

$$\mathbf{R} = \text{r.inf.pos} \cup \text{r.inf.neg} \cup \text{vr.alt.b},$$
$$\emptyset = \text{r.inf.pos} \cap \text{r.inf.neg} \cap \text{vr.alt.b},$$

where influences are over Fragments, r.inf.pos $\subseteq \mathbf{F} \times \mathbf{F}$, r.inf.neg $\subseteq \mathbf{F} \times \mathbf{F}$, and vr.alt.b $\subseteq \mathbf{V} \times \mathbf{V}$.

- Value Types are

$$\mathbf{T} = \{\text{v.Satisfaction}, \text{v.Importance}, \text{v.ProgrStatus}, \text{v.ImplTime}\}.$$

v.Satisfaction = $\{1,0\}$ and 1 reads "satisfied", 0 "not satisfied". v.Importance = $\{1,0\}$ and 1 reads "mandatory", 0 "not mandatory". v.ProgrStatus is called "progress status", and its values are

$$\text{v.ProgrStatus} = \{\textit{none}, \textit{DesignApproved}, \textit{EstimateDone}, \textit{InRoadmap}, \textit{TestReady}, \textit{ApprovedForRelease}\}.$$

v.ImplTime takes a real positive value, v.ImplTime $\in \mathbb{R}^+$.

- Value Assignments are ternary relations over Fragments or relation instances, Value Types, and values of Value Types:

$$\mathbf{V} \subseteq (\mathbf{F} \cup \mathbf{R}) \times \mathbf{T} \times \bigcup_{\text{v.t} \in \mathbf{T}} \text{v.t.}$$

and is partitioned as follows

$$\begin{aligned}\mathbf{V} = {} & (\mathbf{F} \cup \mathbf{R}) \times \{\text{v.Satisfaction}\} \times \{1,0\} \\ & \cup \mathbf{F} \times \{\text{v.Importance}\} \times \{1,0\} \\ & \cup \mathbf{F} \times \{\text{v.ProgrStatus}\} \times \text{v.ProgrStatus} \\ & \cup \mathbf{F} \times \{\text{v.ImplTime}\} \times \text{v.ImplTime},\end{aligned}$$

whereby the intersection of the sets above is empty. Note from the above, that satisfaction values can be assigned to Fragments and relation instances, while values of all other Value Types can be assigned only to Fragments.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$A ::= x \mid y \mid z \mid \ldots$$
$$B ::= r(A) \mid k(A) \mid s(A)$$
$$C ::= B \text{ influences+ } B$$
$$D ::= B \text{ influences- } B$$
$$G ::= \langle A, E, F \rangle$$
$$H ::= G \text{ alternativeTo } G$$
$$Z ::= B \mid C \mid D \mid G \mid H$$

Mapping

Symbols map to domain elements as follows:

- A symbols denote Fragments, $\mathscr{D}(A) \in$ **F**.
- B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s.
- C and D symbols denote, respectively, positive and negative influence relations.
- E symbols denote Value Types, $\mathscr{D}(E) \in$ **T**.
- F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction.
- G symbols denote Value Assignments, $\mathscr{D}(G) \in$ **V**.
- H symbols denote Alternatives, $\mathscr{D}(H) \in$ vr.alt.b.

Language Services

Those of relations and functions in the language.

Figure 12.11 shows a model in L.Avior. Not all Fragments have Value Assignment for all Value Types. For example, AutoAmbList has no satisfaction value, has no importance value, is in the roadmap, and the estimated implementation time is 25 man-days.

To deliver s.AccTime, you can define a function which would work as follows, on a L.Avior model M:

1. Find all Complete v.Satisfaction Outcomes of M, which include a Pick which you are interested in. Let O be the set of all these Outcomes, and $o_i \in O$ a member of that set. An Outcome is Complete for v.Satisfaction if a satisfaction value is assigned to every Fragment and relation instance, which it can be assigned to in a model, according to the language being used.

   Example 12.4.1. Let the Pick be

   $$\mathsf{P} = \{\langle \mathsf{AmbArrIncLoc}, \mathsf{v.Satisfaction}, 1\rangle, \langle \mathsf{AmbArrIncLoc}, \mathsf{v.Importance}, 1\rangle\}.$$

   The Pick is shown in Figure 12.11. Figures 12.7–12.10 show all four Complete v.Satisfaction Outcomes for the Pick above. Although these are models in L.Pollux, they are also models in L.Avior, since the latter was made by adding new Value Types to the former, and without changing functions or syntax in the former. •

2. For each $o_i \in O$, if a Fragment x is such that

   (a) $\langle x, \mathsf{v.Satisfaction}, 1\rangle \in o_i$,

   (b) x is a specification Fragment, $x \in \mathsf{c.s}$,

   (c) x is a leaf Fragment, that is, there are no positive and negative influence relation instances which target x in M,

   (d) there is $\langle x, \mathsf{v.ProgrStatus}, \mathit{InRoadmap}\rangle$, and

   (e) there is $\langle x, \mathsf{v.ImplTime}, v\rangle$, such that $v > 0$.

   then add x to the set T_{o_i}.

Example 12.4.2. *(Example 12.4.1 continued.)* Let o_1 be the Outcome in Figure 12.12, so that

$$T_{o_1} = \{\mathsf{AutoAmbList}, \mathsf{UpdAutoAmbList}, \mathsf{DispSoftwRnkAmb}, \mathsf{DispAmbRnk}\}.$$

•

3. The answer to s.AccTime in M is given for each Complete v.Satisfaction Outcome o_i, by summing the v.ImplTime values assigned to every member of T_{o_i}.

   Example 12.4.3. *(Example 12.4.2 continued.)* For o_1, the total estimated implementation time is

   $$25MD + 4MD + 30MD + 5MD = 64MD.$$

   •

12.5 Summary on Alternatives

A language that aims to support design may need to represent alternative design options in models, via Alternatives and Outcomes. 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 mutually exclusive pairs of Value Assignments. and how to find Outcomes which include no mutually exclusive pair of Value Assignments.

Mutual exclusion was discussed in a limited way in this section. It remains unclear, for example, how to show that ranges of values that can be assigned to a Fragment are mutually exclusive to ranges of values that can be assigned to another Fragment. I will show in the next section that mutual exclusion, as discussed here, is one kind of constraint over Value Assignments, and that there are others which can be relevant to show in models.

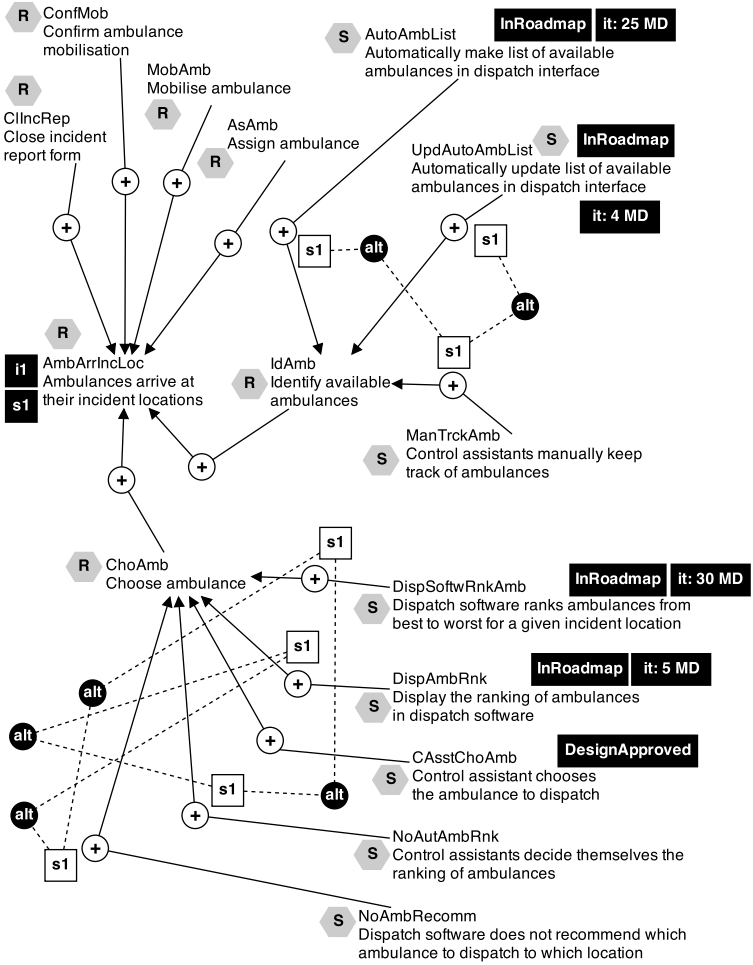


Figure 12.11: A model in L.Avior.

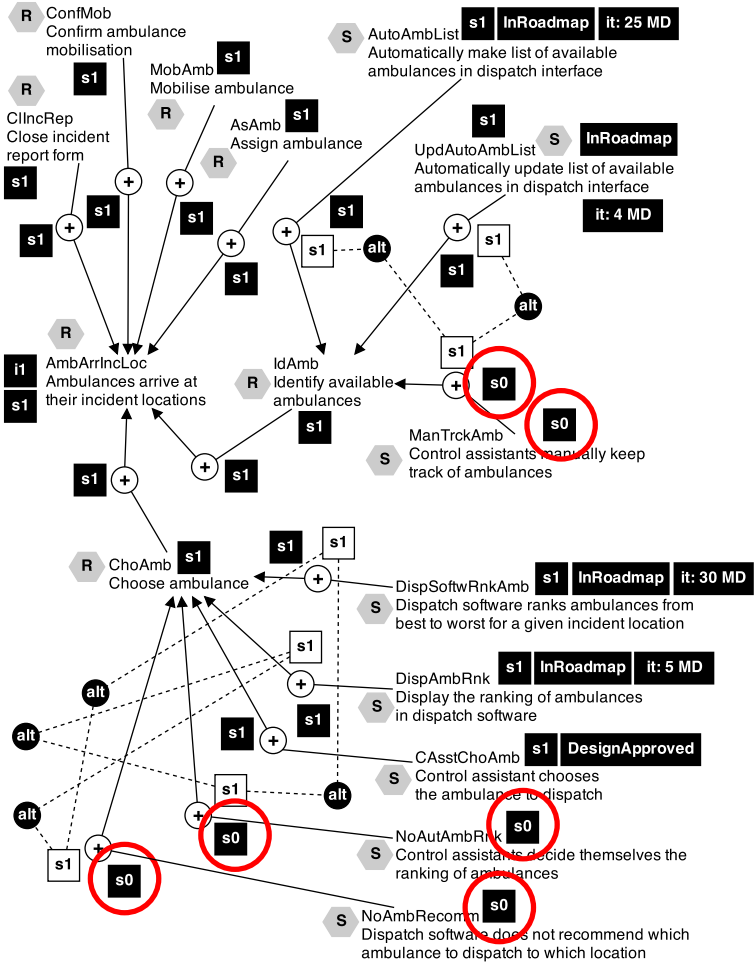


Figure 12.12: A model in L.Avior based on the model in Figure 12.7.

Chapter 13

Constraints

This Chapter focuses on how to have models with Constraints over Value Assignments, which are richer than mutual exclusion from Chapter 12. Here are some Constraints other than mutual exclusion:

- *If x and z are both satisfied, then the implementation cost of y should double.*
- *If x is not acceptable, then estimated revenue from y will decrease by 30%.*
- *If the decision about the acceptability of x arrives after a given date, then implementation cost of x will increase by 50%.*
- *If the implementation time of x is above a, then implementation time of y cannot be over b.*

What is common to the above, is that there is an interdependence between Value Assignments, and that interdependence cannot be reduced, or when it can, it is inefficient to reduce it to mutual exclusion between pairs of Value Assignments. Given such Constraints, the aim is to find one or all Outcomes which satisfy them. The overall idea is that you need to define a system of equations which takes into account all constraints in a model, and find all Outcomes which are its solutions. The rest of this Chapter looks at the following questions:

- *How to represent various kinds of constraints? (Section 13.1)*
- *How to find Outcomes when the model has Constraints in a model? (Section 13.2)*

13.1 Representing Constraints

Constraints define n-ary relations over Value Assignments. The domain of the language needs to reflect the properties of the Constraints which are allowed in the language, and the syntax should specify the format in which these Constraints are written.

While it was straightforward to represent in graphs that two Value Assignments are mutually exclusive, there is no corresponding simple representation of many other kinds of Constraints.

Consider a language in which a model M is a set of Z symbols, where each Z symbol is generated according to the following BNF rules:

$$
\begin{aligned}
A &::= x \mid y \mid z \mid \ldots \\
B &::= r(A) \mid k(A) \mid s(A) \\
C &::= B \text{ influences+ } B \\
D &::= B \text{ influences- } B \\
G &::= \langle A, E, F \rangle \\
H &::= G \text{ alternativeTo } G \\
I &::= A.E \\
J &::= G \mid I \mid N \\
K &::= J \mid K + K \mid K - K \mid K * K \mid K / K \\
L &::= K = K \mid K > K \mid K < K \mid K \geq K \mid K \leq K \\
Z &::= B \mid C \mid D \mid G \mid H \mid L
\end{aligned}
$$

Above, I symbols are variables without an assigned value, as opposed to G symbols where the F symbol stands for the assigned value. A N symbol is a real number. K symbols reflect the arithmetic operations, and L symbols are used to write equality and inequality. L symbols are symbols for Constraints.

Suppose that this language is such that it can represent all that L.Avior could, so that the following Constraint can be associated to the L.Avior model in Figure 12.11.

$$\text{ChoAmb.v.ImplTime} \leq 100$$

and if it included in a Pick, then you are looking for Outcomes which include a Value Assignment which assigns at most 100 v.ImplTime value to ChoAmb. You may be assuming or computing the v.ImplTime

value of ChoAmb, but in any case, you will not be interested in Outcomes where the assigned value is greater than 100.

When reading Constraints such as above, recall the notational convention that the pair ChoAmb and v.ImplTime defines a variable, which can be written

$$\text{ChoAmb.v.ImplTime}$$

and if it is assigned a value, say 50, then there are two equivalent ways of writing that Value Assignment, namely

$$\text{ChoAmb.v.ImplTime} = 50 \equiv \langle \text{ChoAmb}, \text{v.ImplTime}, 50 \rangle.$$

To illustrate both points above, I define a new language which allows you to write constraints using arithmetic operations over Value Assignments in L.Avior models.

There can be more complicated Constraints, such as

$$\text{AutoAmbList.v.ImplTime} * \text{AutoAmbList.v.Satisfaction} < 20$$

which says that I am interested in Outcomes where, if AutoAmbList is satisfied, then its implementation time has to be below 20. Note that this Constraint will be satisfied by Outcomes where AutoAmbList is not satisfied, since v.Satisfaction is binary. But it will not be satisfied by Outcomes where it is satisfied, and implementation time is equal to or above 20.

I could add the following Constraint to aPick, to say that I am interested in Outcomes in which implementing AutoAmbList takes at most twice the time to implement UpdAutoAmbList, if both are satisfied:

$$\text{AutoAmbList.v.Satisfaction} * \text{AutoAmbList.v.ImplTime}$$
$$\leq 2 * \text{UpdAutoAmbList.v.Satisfaction} * \text{UpdAutoAmbList.v.ImpltTime}.$$

Constraints have to be sensitive to the Value Type, as arithmetic operations do not need to make sense for any Value Type. For example, multiplying two values of v.ProgrStatus makes no sense if multiplication is used as in arithmetic. You could define new operators for combining such values, but they would not be those of arithmetic.

Note that r.alt.b, when defined over Value Assignments of Value Types for which arithmetic operations make sense, can be rewritten

as a Constraint. I already did this when I rewrote r.alt.b instances in Figures 12.7–12.10 in a system of equations in Section 12.3.

Below is a definition of the language L.Alphard, which has the syntax given earlier. Its domain is loosely defined, as it includes no detailed definition of arithmetic operations.

L.Alphard is made by extending L.Avior. Aside from differences in syntax, the domains are also different. In L.Alphard, the domain includes Constraints and real numbers $\mathbb{R}$. A Constraint in the domain is a set of relation instances over Value Assignments. Each of these relation instances is such that when the variables in the Constraint obtain values given in the Value Assignments in the relation instance, the arithmetic expression of the Constraint is correct. To clarify this, consider the following Constraint:

$$\mathsf{ChoAmb.v.ImplTime} \leq 100.$$

Recall that v.ImplTime is the set of positive real numbers. It follows that any positive real which is at most 100, that is, any real in $[0,100]$, makes the arithmetic expression above correct. This Constraint is thus a set of instances of a unary relation, and that set is

$$\{\, \langle \mathsf{ChoAmb}, \mathsf{v.ImplTime}, v\rangle \mid v \in [0,100] \text{ and } v \in \mathbb{R} \,\}.$$

Consider this Constraint for another illustration

$$\begin{aligned}&\mathsf{AutoAmbList.v.Satisfaction} * \mathsf{AutoAmbList.v.ImpltTime}\\ &\leq 2 * \mathsf{UpdAutoAmbList.v.Satisfaction} * \mathsf{UpdAutoAmbList.v.ImpltTime}.\end{aligned}$$

In this case, the Constraint defines a set of instances of a four-place relation, over assignments of values to these four variables

$$\begin{aligned}&\mathsf{AutoAmbListv.Satisfaction}, \mathsf{AutoAmbListv.ImplTime},\\ &\mathsf{UpdAutoAmbListv.Satisfaction}, \mathsf{UpdAutoAmbList.v.ImpltTime}.\end{aligned}$$

That set of instances are all assignments of values to all four variables above, such that the arithmetic expression in the Constraint is correct. The following is an instance in that set

$$\begin{aligned}&(\, \langle \mathsf{AutoAmbList}, \mathsf{v.Satisfaction}, 0\rangle, \langle \mathsf{AutoAmbList}, \mathsf{v.ImplTime}, 50\rangle,\\ &\langle \mathsf{UpdAutoAmbList}, \mathsf{v.Satisfaction}, 1\rangle,\\ &\langle \mathsf{UpdAutoAmbList}, \mathsf{v.ImplTime}, 10\rangle \,).\end{aligned}$$

Language: Alphard

Language Modules L.Alphard

r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, vr.alt.b, f.sat.inf.pos, f.sat.inf.neg, f.sat.alt.b, f.sat.leaf, f.imp.asm, f.chk.progrstatus

Domain

- The domain is made of a set of Fragments **F**, relation instances **R**, Value Types **T**, Value Assignments **V**, Constraints **Q**, and real numbers $\mathbb{R}$.
- Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \mathsf{c.r} \cup \mathsf{c.k} \cup \mathsf{c.s}$ and $\mathsf{c.r} \cap \mathsf{c.k} \cap \mathsf{c.s} = \emptyset$.
- Relation instances are over Fragments or Value Assignments, $\mathbf{R} = (\mathbf{F} \times \mathbf{F}) \cup \mathbf{V}^n$, and are partitioned as follows:

$$\mathbf{R} = \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg} \cup \mathsf{vr.alt.b} \cup \bigcup \mathbf{Q},$$
$$\emptyset = \mathsf{r.inf.pos} \cap \mathsf{r.inf.neg} \cap \mathsf{vr.alt.b} \cap \bigcap \mathbf{Q},$$

  where influences are over Fragments, $\mathsf{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\mathsf{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$, and $\mathsf{vr.alt.b} \subseteq \mathbf{V} \times \mathbf{V}$. Each Constraint is a n-ary relation over Value Assignments so that $\mathbf{Q} \subseteq \wp(\mathbf{V}^n)$.

- Value Types are

$$\mathbf{T} = \{\mathsf{v.Satisfaction}, \mathsf{v.Importance}, \mathsf{v.ProgrStatus}, \mathsf{v.ImplTime}\}.$$

  v.Satisfaction = $\{1,0\}$ and 1 reads “satisfied”, 0 “not satisfied”. v.Importance = $\{1,0\}$ and 1 reads “mandatory”, 0 “not mandatory”. v.ProgrStatus is called “progress status”,

and its values are

$$\begin{aligned}\text{v.ProgrStatus} = \{&\textit{none}, \textit{DesignApproved},\\ &\textit{EstimateDone}, \textit{InRoadmap},\\ &\textit{TestReady}, \textit{ApprovedForRelease}\}.\end{aligned}$$

v.ImplTime takes a real positive value, $\text{v.ImplTime} \in \mathbb{R}^+$.

- Value Assignments are ternary relations over Fragments or relation instances, Value Types, and values of Value Types:

$$\mathbf{V} \subseteq (\mathbf{F} \cup \mathbf{R}) \times \mathbf{T} \times \bigcup_{\text{v.t} \in \mathbf{T}} \text{v.t.}$$

and is partitioned as follows

$$\begin{aligned}\mathbf{V} = &(\mathbf{F} \cup \mathbf{R}) \times \{\text{v.Satisfaction}\} \times \{1,0\}\\ &\cup \mathbf{F} \times \{\text{v.Importance}\} \times \{1,0\}\\ &\cup \mathbf{F} \times \{\text{v.ProgrStatus}\} \times \text{v.ProgrStatus}\\ &\cup \mathbf{F} \times \{\text{v.ImplTime}\} \times \text{v.ImplTime},\end{aligned}$$

whereby the intersection of the sets above is empty. Note from the above, that satisfaction values can be assigned to Fragments and relation instances, while values of all other Value Types can be assigned only to Fragments.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$,

where every ϕ is generated according to the following BNF rules:

$$
\begin{aligned}
A &::= x \mid y \mid z \mid \ldots \\
B &::= r(A) \mid k(A) \mid s(A) \\
C &::= B \text{ influences+ } B \\
D &::= B \text{ influences- } B \\
G &::= \langle A, E, F \rangle \\
H &::= G \text{ alternativeTo } G \\
I &::= A.E \\
J &::= G \mid I \mid N \\
K &::= J \mid K + K \mid K - K \mid K * K \mid K / K \\
L &::= K = K \mid K > K \mid K < K \mid K \geq K \mid K \leq K \\
Z &::= B \mid C \mid D \mid G \mid H \mid L
\end{aligned}
$$

Mapping

Symbols map to domain elements as follows:

- A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$.
- B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s.
- C and D symbols denote, respectively, positive and negative influence relations.
- E symbols denote Value Types, $\mathscr{D}(E) \in \mathbf{T}$.
- F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction.
- G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$.
- H symbols denote Alternatives, $\mathscr{D}(H) \in$ vr.alt.b.
- I symbols denote variables, that is, pairs of Fragment or relation instance, and a Value Type, $\mathscr{D}(I) \in (\mathbf{F} \cup \mathbf{R}) \times \mathbf{T}$.
- N symbols denote a real number, $\mathscr{D}(N) \in \mathbb{R}$.

- $K + K$ denotes the sum of K and K,

$$\mathscr{D}(K + K) = \mathscr{D}(K) + \mathscr{D}(K),$$

- $K - K$ denotes the subtraction of K on the right side of "-" from the K on the left side of "-",

$$\mathscr{D}(K - K) = \mathscr{D}(K) - \mathscr{D}(K),$$

- $K * K$ denotes the result of multiplying K and K,

$$\mathscr{D}(K * K) = \mathscr{D}(K) * \mathscr{D}(K),$$

- K / K denotes the result of dividing the K on the left side of "/" with the K on the right side of "/",

$$\mathscr{D}(K / K) = \mathscr{D}(K) / \mathscr{D}(K),$$

- $K = K$ denotes that the two K are equal,
- $K > K$ denotes that the K on the left side of ">" is greater than the K on the right side of ">",
- $K < K$ denotes that the K on the left side of "<" is smaller than the K on the right side of "<",
- $K \geq K$ denotes that the K on the left side of "≥" is equal or greater than the K on the right side of "≥", and
- $K \leq K$ denotes that the K on the left side of "≤" is equal or smaller than the K on the right side of "≤".

Language Services

Those of relations and functions in the language.

Observe how the mapping works in L.Alphard. The arithmetic operations map to standard arithmetic operations, so that when I write

$$\mathscr{D}(K + K) = \mathscr{D}(K) + \mathscr{D}(K),$$

the aim of the formula is to show that the "+" symbol should work the same way that addition works in arithmetic, and this is shown by simply taking it out of $\mathscr{D}$. The mapping works in this same way for the rest of the arithmetic operators, equality, and inequality symbols.

13.2 Constraints in Outcome Search

The aim now is to show a way to find Outcomes in a model with Constraints. The idea is the same as in Section 12.3: define a Pick, convert the model in a system of equations such that any solution of that system is an Outcome which includes the Pick. As I will illustrate below, the only change relative to Section 12.3 is that the system of equations now includes also all of the Constraints in the model.

I will proceed step by step to define the system of equations from the model in L.Alphard, shown in Figure 13.1. The Figure does not include Constraints, and I will introduce them in due time below.

1. I am interested in finding all Complete and Coherent Outcomes of the model in Figure 13.1, which include the following Pick

$$\begin{aligned}\mathsf{P} = \{\, \langle x, \mathsf{v.Satisfaction}, 1\rangle \mid \forall\, x \in M \text{ s.t.} \\ x \in \mathbf{F} \text{ and } \langle x, \mathsf{v.Importance}, 1\rangle \\ \text{and } \langle x, \mathsf{v.ProgrStatus}, \mathit{DesignApproved}\rangle \,\}.\end{aligned}$$

2. Observe that the model in Figure 13.1 has Value Assignments of four Value Types, v.Satisfaction, v.Importance, v.ImplTime, and v.ProgrStatus. It follows that the system of equations will be over four groups of variables, one per Value Type. Below is a sample member of each of these groups

$$\begin{aligned}&\mathsf{AmbArrIncLoc.v.Satisfaction},\\ &\mathsf{AmbArrIncLoc.v.Importance},\\ &\mathsf{DispSoftwRnkAmb.v.ImplTime},\\ &\mathsf{DispAmbRnk.v.ProgrStatus}.\end{aligned}$$

   I now define abbreviations for all variables in each group.

   (a) For each Fragment x in M, I define a variable w_x as the variable which takes the v.Satisfaction value

$$w_x \equiv x.\mathsf{v.Satisfaction}$$

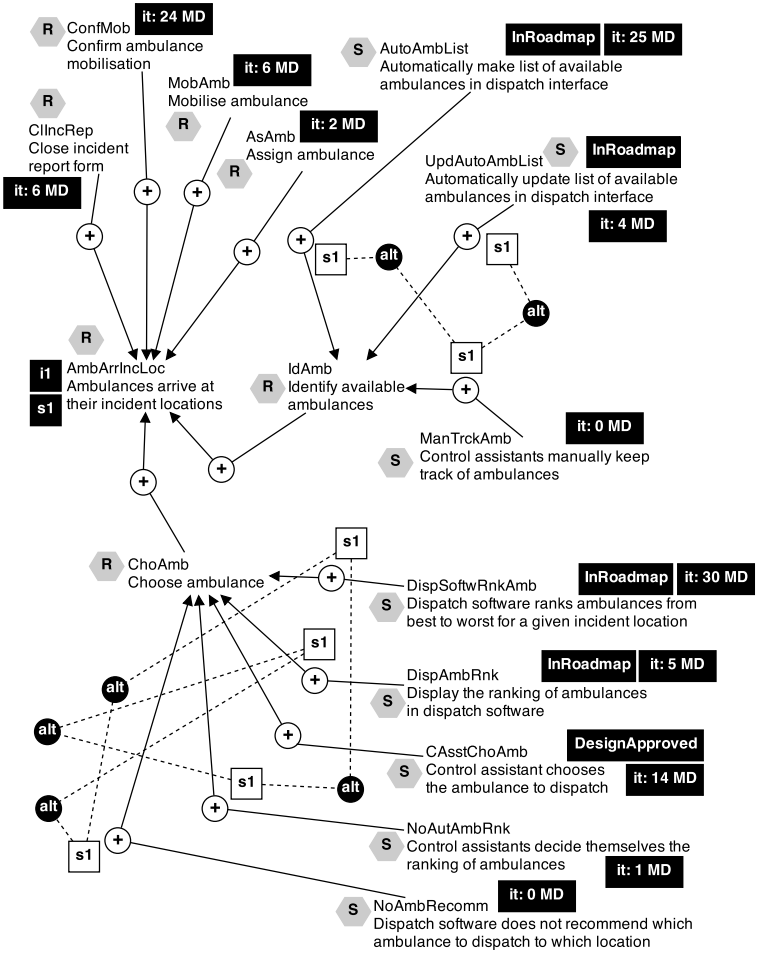


Figure 13.1: A model in L.Alphard.

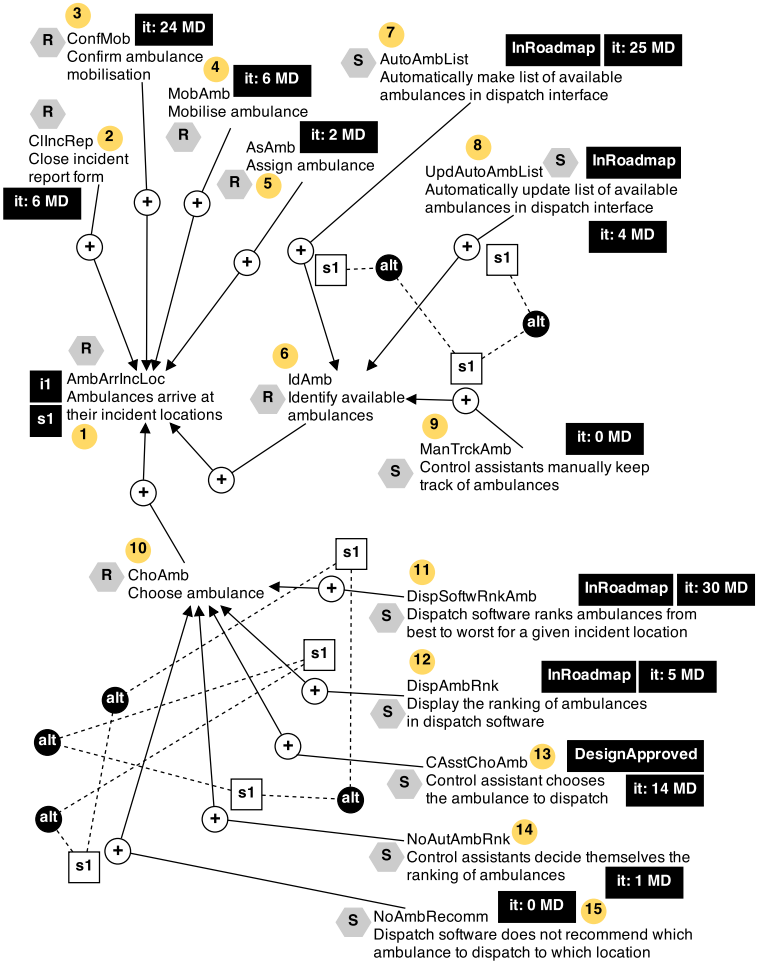


Figure 13.2: Abbreviations of Fragments in variable numbers.

and to further simplify notation, I have these equivalences

$$w_1 \equiv w_{\text{AmbArrIncLoc}}, \quad w_2 \equiv w_{\text{ClIncRep}},$$
$$w_3 \equiv w_{\text{ConfMob}}, \quad w_4 \equiv w_{\text{MobAmb}},$$
$$w_5 \equiv w_{\text{AsAmb}}, \quad w_6 \equiv w_{\text{IdAmb}},$$
$$w_7 \equiv w_{\text{AutoAmbList}}, \quad w_8 \equiv w_{\text{UpdAutoAmbList}},$$
$$w_9 \equiv w_{\text{ManTrckAmb}}, \quad w_{10} \equiv w_{\text{ChoAmb}},$$
$$w_{11} \equiv w_{\text{DispSoftwRnkmb}}, \quad w_{12} \equiv w_{\text{DispAmbRnk}},$$
$$w_{13} \equiv w_{\text{AsstChoAmb}}, \quad w_{14} \equiv w_{\text{NoAutAmbRnk}},$$
$$w_{15} \equiv w_{\text{NoAmbRecomm}}.$$

(b) I then define variables which take values of v.Importance for each Fragment x in M, define a variable w_x as the variable which takes the v.Satisfaction value

$$p_x \equiv x.\text{v.Importance}$$

and I use these equivalences

$$p_1 \equiv p_{\text{AmbArrIncLoc}}, \quad p_2 \equiv p_{\text{ClIncRep}},$$
$$p_3 \equiv p_{\text{ConfMob}}, \quad p_4 \equiv p_{\text{MobAmb}},$$
$$p_5 \equiv p_{\text{AsAmb}}, \quad p_6 \equiv p_{\text{IdAmb}},$$
$$p_7 \equiv p_{\text{AutoAmbList}}, \quad p_8 \equiv p_{\text{UpdAutoAmbList}},$$
$$p_9 \equiv p_{\text{ManTrckAmb}}, \quad p_{10} \equiv p_{\text{ChoAmb}},$$
$$p_{11} \equiv p_{\text{DispSoftwRnkmb}}, \quad p_{12} \equiv p_{\text{DispAmbRnk}},$$
$$p_{13} \equiv p_{\text{AsstChoAmb}}, \quad p_{14} \equiv p_{\text{NoAutAmbRnk}},$$
$$p_{15} \equiv p_{\text{NoAmbRecomm}}.$$

(c) I now add equivalences for variables which take v.ImplTime values, and which have this format

$$t_x \equiv x.\text{v.ImplTime}$$

and I will use the following

$$t_1 \equiv t_{\text{AmbArrIncLoc}}, \quad t_2 \equiv t_{\text{ClIncRep}},$$
$$t_3 \equiv t_{\text{ConfMob}}, \quad t_4 \equiv t_{\text{MobAmb}},$$
$$t_5 \equiv t_{\text{AsAmb}}, \quad t_6 \equiv t_{\text{IdAmb}},$$
$$t_7 \equiv t_{\text{AutoAmbList}}, \quad t_8 \equiv t_{\text{UpdAutoAmbList}},$$
$$t_9 \equiv t_{\text{ManTrckAmb}}, \quad t_{10} \equiv t_{\text{ChoAmb}},$$
$$t_{11} \equiv t_{\text{DispSoftwRnkmb}}, \quad t_{12} \equiv t_{\text{DispAmbRnk}},$$
$$t_{13} \equiv t_{\text{AsstChoAmb}}, \quad t_{14} \equiv t_{\text{NoAutAmbRnk}},$$
$$t_{15} \equiv t_{\text{NoAmbRecomm}}.$$

(d) Finally, as far as variables for Fragments are concerned, there are variables which take v.ProgrStatus values

$$s_x \equiv x.\text{v.ImplTime}$$

and I will use the following

$$s_1 \equiv s_{\text{AmbArrIncLoc}}, \quad s_2 \equiv s_{\text{ClIncRep}},$$
$$s_3 \equiv s_{\text{ConfMob}}, \quad s_4 \equiv s_{\text{MobAmb}},$$
$$s_5 \equiv s_{\text{AsAmb}}, \quad s_6 \equiv s_{\text{IdAmb}},$$
$$s_7 \equiv s_{\text{AutoAmbList}}, \quad s_8 \equiv s_{\text{UpdAutoAmbList}},$$
$$s_9 \equiv s_{\text{ManTrckAmb}}, \quad s_{10} \equiv s_{\text{ChoAmb}},$$
$$s_{11} \equiv s_{\text{DispSoftwRnkmb}}, \quad s_{12} \equiv s_{\text{DispAmbRnk}},$$
$$s_{13} \equiv s_{\text{AsstChoAmb}}, \quad s_{14} \equiv s_{\text{NoAutAmbRnk}},$$
$$s_{15} \equiv s_{\text{NoAmbRecomm}}.$$

Figure 13.2 shows which number in a variable name corresponds to which Fragment in the model in Figure 13.1.

3. L.Alphard lets me assign v.Satisfaction values over relation instances, so that I need variables for all relation instances as well. They will have the following format, where x and y are Fragments

$$w_{x.p.y} \equiv (x \text{ influences+ } y).\text{v.Satisfaction}.$$

Given the other abbreviations defined so far, I will have, for example

$$w_{2.p.1} \equiv (\text{ClIncRep influences+ AmbArrIncLoc}).\text{v.Satisfaction}.$$

I will not write all the other influence relation instances, as it should be clear what they are from the abbreviations above.

4. So far, I have defined the set of variables that will appear in the system of equations. Relation instances and functions for propagating satisfaction values give me a first set of equations. As there are only positive influences in Figure 13.1, and each uses the rules from f.sat.inf.pos, the resulting equations are (same as in Section 12.3)

$$
\begin{aligned}
0 &= w_2 - w_{2.p.1}, & 0 &= w_3 - w_{3.p.1},\\
0 &= w_4 - w_{4.p.1}, & 0 &= w_5 - w_{5.p.1},\\
0 &= w_6 - w_{6.p.1}, & 0 &= w_{10} - w_{10.p.1},\\
0 &= w_7 - w_{7.p.6}, & 0 &= w_8 - w_{8.p.6},\\
0 &= w_9 - w_{9.p.6}, & &\\
0 &= w_{11} - w_{11.p.10}, & 0 &= w_{12} - w_{12.p.10},\\
0 &= w_{13} - w_{13.p.10}, & 0 &= w_{14} - w_{14.p.10},\\
0 &= w_{15} - w_{15.p.10}. & &
\end{aligned}
$$

5. r.alt.b instances also give equations, and they are found the same way as in Section 12.3

$$
\begin{aligned}
1 &= |w_{7.p.6} - w_{9.p.6}|,\\
1 &= |w_{8.p.6} - w_{9.p.6}|,\\
1 &= |w_{11.p.10} - w_{14.p.10}|,\\
1 &= |w_{11.p.10} - w_{15.p.10}|,\\
1 &= |w_{12.p.10} - w_{14.p.10}|,\\
1 &= |w_{12.p.10} - w_{15.p.10}|,\\
w_1 &= w_{2.p.1} \cdot w_{3.p.1} \cdot w_{4.p.1} \cdot w_{5.p.1} \cdot w_{6.p.1} \cdot w_{10.p.1},\\
w_6 &= w_{7.p.6} \cdot w_{8.p.6} + w_{9.p.6},\\
w_{10} &= w_{11.p.10} \cdot w_{12.p.10} \cdot w_{13.p.10}\\
&\quad + w_{13.p.10} \cdot w_{14.p.10} \cdot w_{15.p.10}.
\end{aligned}
$$

6. The first set of Constraints reflects the Value Assignments to leaf Fragments in Figure 13.1. These Constraints are as follows,

for v.ImplTime Value Assignments

$$t_2 = 6, \quad t_3 = 24, \quad t_4 = 6, \quad t_5 = 2,$$
$$t_7 = 25, \quad t_8 = 4, \quad t_9 = 0,$$
$$t_{11} = 30, \quad t_{12} = 5, \quad t_{13} = 14, \quad t_{14} = 1,$$
$$t_{15} = 0,$$

and they are the following, for v.ProgrStatus

$$t_7 = \mathit{InRoadmap}, \quad t_8 = \mathit{InRoadmap},$$
$$t_{11} = \mathit{InRoadmap}, \quad t_{12} = \mathit{InRoadmap},$$
$$t_{13} = \mathit{DesignApproved}.$$

7. Observe that I did not define a specific relation which says how the implementation time of, say ChoAmb, depends on that of other Fragments in the model. I do this with the following Constraints

$$t_1 = \sum_{i \in \{2,3,4,5,6,10\}} t_i * w_i,$$
$$t_6 = \sum_{i \in \{7,8,9\}} t_i * w_i,$$
$$t_{10} = \sum_{i \in \{11,12,13,14,15\}} t_i * w_i.$$

   The Constraints convey the idea that implementation time of a Fragment x is the sum of the implementation times of all Fragments which are satisfied, and which are positively influencing x.

8. I have no Constraints for v.Importance Value Assignments, other than the one in the Pick, which is

$$p_1 = 1.$$

9. The Pick gives two other Constraints, namely

$$w_1 = 1,$$
$$s_1 = \mathit{DesignApproved}.$$

10. Recall from the definition of v.ProgrStatus, that *DesignApproved* is the first progress status that can be assigned to a Fragment, and following f.chkprogrstatus that it can be assigned only to Fragments which have no other progress status value. Observe also that arithmetic operations are not defined over v.ProgrStatus, so that I need to introduce a coding of its values into, say, integers.

    Since the Pick requires AmbArrIncLoc to get the value *DesignApproved*, I introduce the following equivalences

$$\begin{aligned} \textit{none} &\equiv 0, \\ \textit{DesignApproved} &\equiv 1, \\ \textit{EstimateDone} &\equiv 1, \\ \textit{InRoadmap} &\equiv 1, \\ \textit{TestReady} &\equiv 1, \\ \textit{ApprovedForRelease} &\equiv 1. \end{aligned}$$

    The above reflects the fact that design is approved on any Fragment which either has *DesignApproved* value, or has any v.ProgrStatus value which, according to f.chi.progrstatus, can be assigned after *DesignApproved*, that is, after the design has been approved.

    Furthermore, I consider that if a Fragment x is target of positive influence relation instances, then its progress status depends on the progress status of the Fragments in which these positive influences originate. Therefore, I have the following Constraints

$$\begin{aligned} s_1 &= \prod_{i\in\{2,3,4,5,6,10\}} s_i * w_i, \\ s_6 &= \prod_{i\in\{7,8,9\}} s_i * w_i, \\ s_{10} &= \prod_{i\in\{11,12,13,14,15\}} s_i * w_i. \end{aligned}$$

11. I add the following as a final Constraint

$$t_1 \leq 80.$$

12. At this point, I have equations which correspond to the Pick, to all Constraints resulting from the influence and r.alt.b relations in the model, and to all Constraints I added in relation to v.ImplTime and v.ProgrStatus. All these equations constitute the system of equations to solve, in order to find one or more (if there are) Outcomes which include the Pick.

    Observe that such an Outcome cannot satisfy both AutombList and DispSoftwRnkAmb, as this would violate the Constraint that implementation time of AmbArrIncLoc is below 80 *MD*.

13.3 Summary on Constraints

This Chapter illustrated how Constraints over Value Assignments can be represented in models, and how they can be taken into account when searching for Outcomes. The discussion was limited to constraints defined over arithmetic operations only. I did not discuss the computational complexity of finding Outcomes, and this remains outside the scope of this book.

An important idea in this Chapter, and in part already illustrated in Chapter 12, is that a language may be such that its models can be rewritten as systems of equations. In other words, you can see such models as representations of the underlying systems of equations. This is relevant when you are interested in finding Outcomes which include a specific Pick which you defined. It also follows that you can see languages such as L.Alphard as tools to construct these systems of equations incrementally and iteratively.

Chapter 14

Preferences

When there are several Outcomes, all of which include a Pick you are interested in, which one of these Outcomes do you choose? How do you choose it? Which one is the "best"? What tells you, in a model, if an Outcome is "better" than another? This Chapter 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 14.2)*
2. *How to represent preferences and criteria? (Section 14.3)*
3. *Where to find Criteria in requirements? (Section 14.4)*
4. *How to use preferences to find best Outcomes in models? (Section 14.5)*

14.1 Motivation

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 incident reports in paper format. Some Outcomes will suggest to use dispatching software to manage incident reports, and will, with regards to how incident reports are managed, be more desirable to other Outcomes, which recommend otherwise.

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 compare Outcomes. This involves various activities, such as eliciting preferences, finding criteria, inferring missing preferences and orders over Outcomes.

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 are also major reasons 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.

14.2 Preferences and Criteria Basics

There is considerable research on preferences in philosophy [119, 66], economics [89, 140, 142, 148, 22, 136, 106, 101], operations research [50, 58, 48], and artificial intelligence [6, 44, 42]. In Section 14.2.1, I recall common ideas about two core preference relations, called strict preference and indifference. In Sections 14.2.2, I introduce the preference-related terminology specific to this book, and in Section 14.2.3 I introduce Criteria and relate them to preferences.

14.2.1 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, that neither is better than the other. This is a case of being *indifferent* between 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

Indifference

Strict preference and indifference are two core preference relations [67], 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 book.

Completeness

Completeness is a desirable property when you want to establish a total order over Outcomes. 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 [25], 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.

Comparability

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.

14.2.2 Domains of Preference Relations

In this book, 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.

14.2.3 Criteria

In this book, a preference relation is always associated to a Criterion. A Criterion c, denoted crit.c, is a function over value assignments, such that if there is a preference relation instance

Criterion

$$(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle),$$

and it is associated to crit.c, then

$$\text{crit.c}(\langle x_i, t_j, v_k\rangle) > \text{crit.c}(\langle x_l, t_p, v_q\rangle)$$

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 other 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

$$(\langle x_i, t_j, v_k\rangle, \langle x_l, t_p, v_q\rangle) \in \text{r.Pref.c}$$

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.

14.3 Representing Preferences

A preference relation instance compares two Value Assignments for desirability. To represent the core preference relations from Section 14.2, a language needs relations of two kinds, namely strict preference and indifference. There needs to be one strict preference relation, and one indifference relation per Criterion in the language.

Relation: pref.c	
Strict Preference	r.pref.c

Domain & Dimension

r.pref.c $\subseteq$ **V** $\times$ **V**, where c is a Criterion, and **V** is a set of Value Assignments.

Properties

Irreflexive, antisymmetric, and transitive.

Reading

$(v, w) \in$ vr.pref.c reads "Value Assignment v is strictly more desirable than Value Assignment w on the Criterion c".

Language Services

- **s.IsPref**: Is Value Assignment v strictly preferred to Value Assignment w on crit.c? : Yes, if $(v, w) \in$ vr.pref.c. s.IsPref

L.Bellartrix, the language defined below, adds vr.pref.c and Criteria to L.Pollux. An additional change is that besides v.Satisfaction and v.Importance, the language has $n > 1$ additional Value Types. Each of these is the set of positive reals, $\mathbb{R}^+$. The language gives generic names to each of these Value Types, and you can define aliases for them, which are meaningful in the Problem instance you are solving. I will illustrate this after the language definition.

Language: Bellatrix

Language Modules L.Bellatrix
r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, vr.alt.b, vr.pref.c, f.sat.inf.pos, f.sat.inf.neg, f.sat.alt.b, f.sat.leaf, f.imp.asm

Domain

- The domain is made of a set of Fragments **F**, relation instances **R** with a special subset **P** of preference relation

instances, Value Types **T**, Value Assignments **V**, Criteria **C**, and real numbers $\mathbb{R}$.

- Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \mathsf{c.r} \cup \mathsf{c.k} \cup \mathsf{c.s}$ and $\mathsf{c.r} \cap \mathsf{c.k} \cap \mathsf{c.s} = \emptyset$.

- Relation instances are over Fragments or Value Assignments, $\mathbf{R} = (\mathbf{F} \times \mathbf{F}) \cup (\mathbf{V} \times \mathbf{V})$, and are partitioned as follows:

$$\mathbf{R} = \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg} \cup \mathsf{vr.alt.b} \cup \mathbf{P},$$
$$\emptyset = \mathsf{r.inf.pos} \cap \mathsf{r.inf.neg} \cap \mathsf{vr.alt.b} \cap \mathbf{P},$$

  where

$$\mathbf{P} = \bigcup_{\mathsf{crit.c} \in \mathbf{C}} \mathsf{vr.pref.c},$$

  influences are over Fragments, $\mathsf{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\mathsf{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$, while mutual exclusion and preferences are over Value Assignments $\mathsf{vr.alt.b} \cup \mathbf{P} \subseteq \mathbf{V} \times \mathbf{V}$.

- Value Types are

$$\mathbf{T} = \{\mathsf{v.Satisfaction}, \mathsf{v.Importance}\} \cup \bigcup_{\mathsf{q}=1}^{n>1} \mathsf{v.q}.$$

  v.Satisfaction and v.Importance are binary. Satisfaction value 1 reads "satisfied", and 0 "not satisfied". Importance value 1 reads "mandatory", 0 "not mandatory". Each of the $\mathsf{q} = 1, \ldots, n$ Value Types v.q is the set of positive reals.

- Value Assignments are ternary relations over Fragments or relation instances, Value Types, and values of Value Types:

$$\mathbf{V} \subseteq (\mathbf{F} \cup \mathbf{R}) \times \mathbf{T} \times \bigcup_{\mathsf{v.t} \in \mathbf{T}} \mathsf{v.t.}$$

  and has the following three partitions

$$(\mathbf{F} \cup \mathbf{R}) \times \{\mathsf{v.Satisfaction}\} \times \{1, 0\},$$
$$\mathbf{F} \times \{\mathsf{v.Importance}\} \times \{1, 0\}, \text{ and}$$
$$\mathbf{F} \times \bigcup_{\mathsf{q}=1}^{n>1} \mathsf{v.q} \times \mathbb{R}^{+}.$$

Satisfaction values can be assigned to Fragments and relation instances, while values of all other Value Types can be assigned only to Fragments.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$A ::= x \mid y \mid z \mid \ldots$$
$$B ::= r(A) \mid k(A) \mid s(A)$$
$$C ::= B \text{ influences+ } B$$
$$D ::= B \text{ influences- } B$$
$$G ::= \langle A, E, F \rangle$$

$$H ::= G \text{ alternativeTo } G$$
$$I ::= G \text{ Pref.}J\ G$$
$$J ::= c_1 \mid c_2 \mid \ldots$$
$$Z ::= B \mid C \mid D \mid G \mid H \mid I$$

Mapping

Symbols map to domain elements as follows:

- A symbols denote Fragments, $\mathscr{D}(A) \in$ **F**.
- B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s.
- C and D symbols denote, respectively, positive and negative influence relations.
- E symbols denote Value Types, $\mathscr{D}(E) \in$ **T**.
- F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction.

- G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$.
- H symbols denote Alternatives, $\mathscr{D}(H) \in$ vr.alt.b.
- I symbols denote instances of preference relations, one per Criterion c,

$$\mathscr{D}(I) \in \bigcup_{\text{crit.c} \in \mathbf{C}} \text{vr.pref.c.}$$

- J symbols denote Criteria, $\mathscr{D}(J) \in \mathbf{C}$.

Language Services

Those of relations and functions in the language.

Figure 14.1 is a visualisation of twelve preference relation instances, over a model in L.Bellatrix. The model takes its contents from parts of the model in Figure 13.2. The numbers used to abbreviate variables in Figure 13.2 are the same in Figure 14.1.

Figure 14.1 shows two Criteria, and each is associated to four preferences. crit.AdvDecSup says that the software should provide more advanced decision support. It follows that having the software automatically update the list of available ambulances, and having it make that list is strictly preferred to having control assistants perform this work manually. This is conveyed by the following preferences

⟨UpdAutoAmbList, v.Satisfaction, 1⟩ Pref.AdvDecSup
⟨ManTrckAmb, v.Satisfaction, 1⟩ and
⟨AutoAmbList, v.Satisfaction, 1⟩ Pref.AdvDecSup
⟨ManTrckAmb, v.Satisfaction, 1⟩.

Following the same Criterion, it is more desirable to have the software rank candidate ambulances for dispatching to an incident location, and that it displays the ranking, than not to assist control

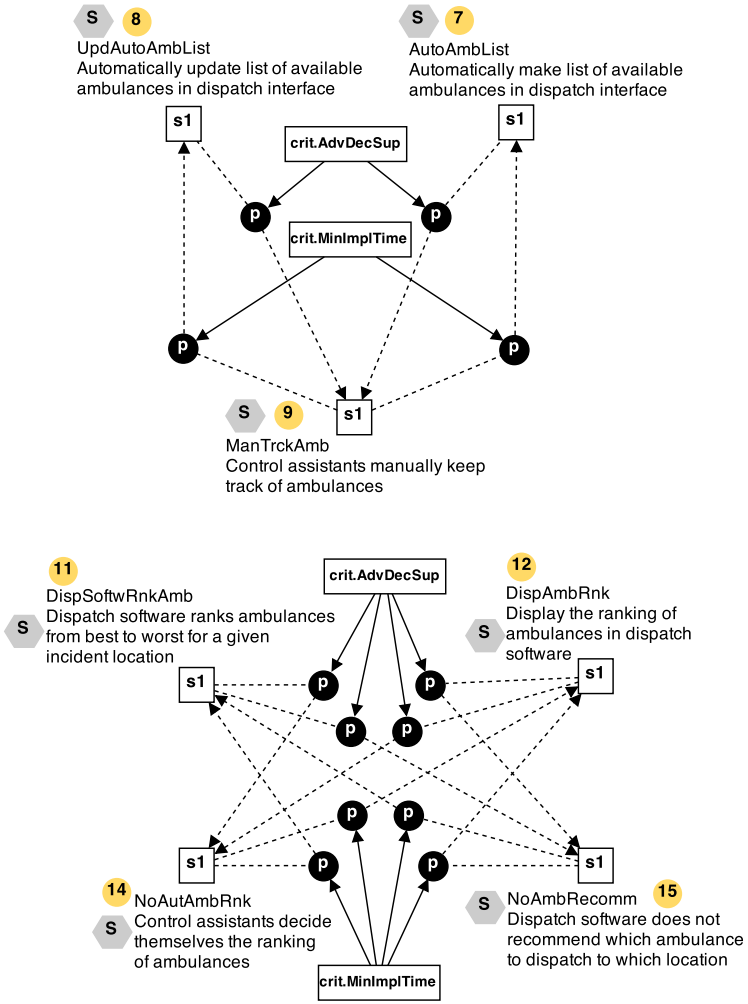


Figure 14.1: Preferences and Criteria in a model in L.Bellatrix.

assistants in this way. This is represented with these preferences

⟨DispSoftwRnkAmb, v.Satisfaction, 1⟩ Pref.AdvDecSup
⟨NoAutAmbRnk, v.Satisfaction, 1⟩,
⟨DispSoftwRnkAmb, v.Satisfaction, 1⟩ Pref.AdvDecSup
⟨NoAmbRecomm, v.Satisfaction, 1⟩,
⟨DispAmbRnk, v.Satisfaction, 1⟩ Pref.AdvDecSup
⟨NoAutAmbRnk, v.Satisfaction, 1⟩, and
⟨DispAmbRnk, v.Satisfaction, 1⟩ Pref.AdvDecSup
⟨NoAmbRecomm, v.Satisfaction, 1⟩.

The second Criterion in Figure 14.1 is crit.MinImplTime, and says that lower implementation times, that is, lower values of v.ImplTime are strictly preferred to higher values of this Value Type. This gives the following preferences

⟨ManTrckAmb, v.Satisfaction, 1⟩ Pref.MinImplTime
⟨UpdAutoAmbList, v.Satisfaction, 1⟩,
⟨ManTrckAmb, v.Satisfaction, 1⟩ Pref.MinImplTime
⟨AutoAmbList, v.Satisfaction, 1⟩,
⟨NoAutoAmbRnk, v.Satisfaction, 1⟩ Pref.MinImplTime
⟨DispSoftwRnkAmb, v.Satisfaction, 1⟩,

⟨NoAutoAmbRnk, v.Satisfaction, 1⟩ Pref.MinImplTime
⟨DispAmbRnk, v.Satisfaction, 1⟩,
⟨NoAmbRecomm, v.Satisfaction, 1⟩ Pref.MinImplTime
⟨DispSoftwRnkAmb, v.Satisfaction, 1⟩, and
⟨NoAmbRecomm, v.Satisfaction, 1⟩ Pref.MinImplTime
⟨DispAmbRnk, v.Satisfaction, 1⟩.

As a digression, note that all the preferences above can be written with variable abbreviations from Section 13.2. These are two examples:

$(w_8 = 1)$ Pref.AdvDecSup $(w_9 = 1)$ and
$(w_7 = 1)$ Pref.AdvDecSup $(w_9 = 1)$.

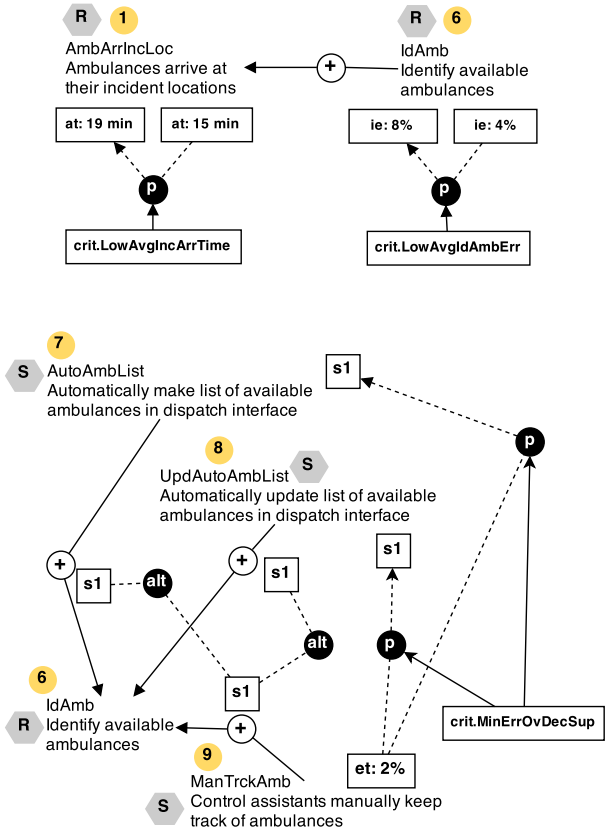


Figure 14.2: More preferences and Criteria in a model in L.Bellatrix.

All the preferences above are strict preferences over Value Assignments *of the same Value Type*, v.Satisfaction. There can also be preferences over Value Assignments over different Value Types, on the same or different Fragments. Figure 14.2 shows examples.

The model part shown in Figure 14.2 involves three new Value Types relative to Figure 14.1. There is v.AvgIncArrTime, which is the average time for an ambulance to arrive at an incident, from the moment the incident was reported. Ideally, this should be minimised, hence the Criterion crit.LowAvgIncArrTime. The Criterion gives this

preference

⟨AmbArrIncLoc, v.LowAvgIncArrTime, 15⟩ Pref.LowAvgIdAmbErr
⟨AmbArrIncLoc, v.AvgIdAmbErr, 19⟩.

v.AvgIdAmbErr is the average percentage of ambulance identifications, in which the ambulance was identified as available, but it in fact was not. That is, this is the average proportion of erroneous identifications of available ambulances. Lower values are preferred, so that there is crit.LowAvgIdAmbErr, which gives this strict preference over two assignments of v.AvgIdAmbErr

⟨IdAmb, v.AvgIdAmbErr, 0.04⟩ Pref.LowAvgIdAmbErr
⟨IdAmb, v.AvgIdAmbErr, 0.08⟩.

Both preferences above are, each, over Value Assignments of values of one Value Type to one Fragment.

There is also the Value Type v.AvgErrTrckAmb, which gives the percentage of errors when keeping track of ambulances manually, as in ManTrckAmb. There is the Criterion crit.MinErrOvDecSup, which says that minimising errors is preferred to giving more decision support to control assistants. Hence these two preferences, which are over Value Assignments of different Value Types, on different Fragments:

⟨ManTrckAmb, v.AvgErrTrckAmb, 0.02⟩ Pref.MinErrOvDecSup
⟨AutoAmbList, v.Satisfaction, 1⟩, and
⟨ManTrckAmb, v.AvgErrTrckAmb, 0.02⟩ Pref.MinErrOvDecSup
⟨UpdAutoAmbList, v.Satisfaction, 1⟩.

Preferences, as in the examples above, suggest how to compare entire Outcomes, depending on the Value Assignments in each. I consider this task in Section 14.5.

14.4 Finding Criteria

Fragments can include statements which suggest many preference relation instances. For example, suppose that stakeholders agree that they prefer lower to higher average times for ambulances to arrive at incidents, and that there is no lower limit to how short the average

time can be. Perhaps a stakeholder said that she wants an ambulance to arrive as quickly as possible to an incident.

Statements such as "low incident response times", "less maintenance", "low cost" have been studied in Requirements Engineering for several decades, as nonfunctional requirements, and as critical for understanding and measuring system quality [16, 108].

I view these statements as suggesting Criteria. For example, "ambulance quickly arrives at incident" gives a Criterion, in that if there are two Alternatives in a model, and they result in different values of a measure of how quickly ambulances arrive at incidents, then the one which gives a better value will be strictly preferred than the other on this Criterion.

A statement such as "ambulance quickly arrives at incident" is not enough by itself to define a Criterion. There should be a Value Type, whose values measure how quickly ambulances arrive at incidents, and it is necessary to clarify what exactly these values measure.

The more general suggestion is to look for Criteria in Fragments which include gradable adjectives, such as quick, slow, big, small, efficient, usable, easy to make, and so on. This is because such adjectives are applied presumably after comparison. Something is fast, because it was compared to something else which seemed less fast. This also means that there is no universal threshold for a gradable adjective to truthfully apply always, everywhere, and in everyone's eyes. You may call a car A fast, because you are used to car B, which is not that fast. Another person may consider A as slow, because she is used to car C, which is faster than A, and so on.

Given a statement with a gradable adjective, such as "ambulance quickly arrives at incident", you can do the following.

1. In the model, identify the Fragments such that Value Assignments on them can be characterised by the statement. These Fragments will talk of things, situations, events, or otherwise, and the Value Assignments will be assigning values that describe some properties of these things, situations, events.

   For example, a Fragment which says that an ambulance does or should arrive at an incident, is about a process of the ambulance arriving to the incident.

2. Define a Value Type, whose values can be used to describe some property of the situation, event, or object mentioned in Fragments in the step above. The property should be such

that the statement with the gradable adjective seems true for some of its values, but not for all of them, and therefore, the value of that property can be used to compare different Value Assignments.

Continuing the example, there can be a Value Type whose values describe the average time for an ambulance to arrive at an incident. If the statement with the gradable adjective is "ambulance quickly arrives at incident", then this statement will apply to some, but not all values of this Value Type. Perhaps 15 min average time for an ambulance to arrive at an incident is quick, but 30 min is not.

3. Make Alternative Value Assignments of the Value Type defined above, and use the statement with the gradable adjective to define strict preferences over these Value Assignments.

   Continuing the example above, if there are two Value Assignments, 15 min and 30 min, then there will necessarily be a strict preference for 15 min over that of 30 min.

Another way to understand the steps above, and using the terminology of the example, is that you are trying to a scale such that, when given any two different values on that scale, you or a stakeholder can say which of the two values describes ambulances arriving faster than the other. The solution in the example is to have a Value Type which gives the average time to arrive at an incident, so that when you have a value v and w of that Value Type, and $v \neq w$, then it is clear that, if $v < w$, then v describes ambulances arriving faster than w.

I define crit.LowAvgIncArrTime below, in such a way as to generate preference relation instances over Value Assignments, when it is given a pair of v.AvgIncArrTime 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. It works as a function. Nevertheless, I want to distinguish Criteria from other kinds of functions, hence the dedicated template.

Criterion: LowAvgIncArrTime

Ambulance quickly arrives at incident crit.LowAvgIncArrTime

Input

A pair of Value Assignments

$\langle x_1, \text{v.AvgIncArrTime}, v_1 \rangle$, and
$\langle x_2, \text{v.AvgIncArrTime}, v_2 \rangle$.

Do

Let v_i be the minimum and v_j the maximum in $\{v_1, v_2\}$.

Output

$(\langle x_i, \text{v.AvgIncArrTime}, v_i \rangle, \langle x_j, \text{v.AvgIncArrTime}, v_j \rangle)$
$\in$ vr.pref.LowAvgIncArrTime.

Language Services

- **s.WhLowAvgIncArrTime**: According to s.WhLowAvgIncArrTime

  crit.LowAvgIncArrTime,

  which of the Value Assignments $\langle x_1, \text{v.AvgIncArrTime}, v_1 \rangle$ and $\langle x_2, \text{v.AvgIncArrTime}, v_2 \rangle$ is strictly preferred to the other? : The one strictly preferred according to the output of this module.

You can define a more general Criterion, to use with Value Assignments where Value Types are real numbers, and lowest or highest values are the most desirable. This Criterion is below.

Criterion: d.t

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 $w = (\langle x_i, \text{v.t}, v_i \rangle, \langle x_j, \text{v.t}, v_j \rangle) \in$ vr.pref.d.t,
- if $d = High$, then $w = (\langle x_j, \text{v.t}, v_j \rangle, \langle x_i, \text{v.t}, v_i \rangle) \in$ vr.pref.d.t.

Output

w.

Language Services

- **s.WhPref.d.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. s.WhPref.d.t

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.

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 14.4.1. 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 [19], 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: low.rev.h

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 = $(\langle x_i, \text{v.t}, v_i\rangle, \langle x_j, \text{v.t}, v_j\rangle) \in$ vr.pref.rev.d.t, else
w = $(\langle x_j, \text{v.t}, v_j\rangle, \langle x_i, \text{v.t}, v_i\rangle) \in$ vr.pref.rev.d.t.

Output

w.

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. s.WhPref.low.rev.h

•

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 [156, 103, 48]. 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.

14.5 Better and Best Outcomes

Given a model which includes Criteria and preference relation instances over Fragments, how would you deliver the following Language Services?

- **s.BestOutcome**: Which is the best Complete Outcome of M? s.BestOutcome
- **s.BetterOutcome**: Which of two Outcomes o_i and o_j is better in model M? s.BetterOutcome

Both are problems of preference aggregation, that is, of taking preference relation instances over Value Assignments in a model, and deciding how to use them together in order to compare Complete Outcomes. Preference aggregation is a topic studied in various domains, including, for example, artificial intelligence [18], formal logic [113], multi-criteria decision-making [48], social choice [106].

I choose one approach to preference aggregation from artificial intelligence. Given a model M with Value Assignments, preference relations, and Criteria, I want to deliver s.BestOutcome and s.BetterOutcome by mapping the Value Assignments and preferences to a Conditional Preference Network, CP-Net hereafter [18]. The Language Services s.BestOutcome and s.BetterOutcome are delivered by applying known algorithms to the resulting CP-Net.

To reduce the number of preference relation instances that need to be elicited, CP-Nets use the *conditional preference relation*.

Conditional preference

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 conditional preferences in models. This is done with the relation vr.pref.cond below.

Relation: pref.cond
Conditional preference r.pref.cond
Domain & Dimension r.pref.cond $\subseteq$ **V** $\times$ **P**, where **V** is a set of Value Assignments, and **P** is a set of vr.pref instances.
Properties If the preference $p \in P$ should be taken into account when comparing Outcomes, all of which include $\langle x, \text{v.t}, v \rangle$, then let $(\langle x, \text{v.t}, v \rangle, p) \in$ vr.pref.cond.

Reading

$(\langle x, \text{v.t}, v\rangle, p) \in$ vr.pref.cond reads "use the preference relation p when comparing Outcomes, only if all these Outcomes s include $\langle x, t, v\rangle$".

Language Services

- **s.IsCondPref**: Should the preference p be used to compare Outcomes in the set O? : Yes, if there is $(\langle x, \text{v.t}, v\rangle, p) \in$ vr.pref.cond and all Outcomes in O include $\langle x, t, v\rangle$.

s.IsCondPref

To illustrate how to make CP-Nets from a model, I define below the language called L.Elnath, by adding vr.pref.cond to L.Bellatrix, and allowing manual assignment of binary v.Approval values on Fragments.

Language: Elnath

Language Modules
r.inf.pos, r.inf.neg, f.map.abrel.g, f.cat.ksr, vr.alt.b, vr.pref.c, vr.pref.cond, f.sat.inf.pos, f.sat.inf.neg, f.sat.alt.b, f.sat.leaf, f.imp.asm

L.Elnath

Domain

- The domain is made of a set of Fragments **F**, non-conditional relation instances **R** with a special subset **P** of preference relation instances, conditional relation instances **cP**, Value Types **T**, Value Assignments **V**, Criteria **C**, and real numbers $\mathbb{R}$.
- Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, **F** = c.r $\cup$ c.k $\cup$ c.s and c.r $\cap$ c.k $\cap$ c.s = $\emptyset$.

- Relation instances are over Fragments or Value Assignments, $\mathbf{R} = (\mathbf{F} \times \mathbf{F}) \cup (\mathbf{V} \times \mathbf{V})$, and are partitioned as follows:

$$\mathbf{R} = \mathsf{r.inf.pos} \cup \mathsf{r.inf.neg} \cup \mathsf{vr.alt.b} \cup \mathbf{P},$$
$$\emptyset = \mathsf{r.inf.pos} \cap \mathsf{r.inf.neg} \cap \mathsf{vr.alt.b} \cap \mathbf{P},$$

  where

$$\mathbf{P} = \bigcup_{\mathsf{crit.c} \in \mathbf{C}} \mathsf{vr.pref.c},$$

  influences are over Fragments, $\mathsf{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\mathsf{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$, while mutual exclusion and non-conditional preferences are over Value Assignments $\mathsf{vr.alt.b} \cup \mathbf{P} \subseteq \mathbf{V} \times \mathbf{V}$.

- Conditional preference relation instances are over Value Assignments and non-conditional preference relation instances in $\mathbf{P}$,

$$\mathbf{cP} = \mathsf{vr.pref.cond},$$
$$\mathsf{vr.pref.cond} \subseteq \mathbf{V} \times \mathbf{P}.$$

- Value Types are

$$\mathbf{T} = \{\mathsf{v.Satisfaction}, \mathsf{v.Importance}, \mathsf{v.Approval}\} \cup \bigcup_{\mathsf{q}=1}^{n>1} \mathsf{v.q}.$$

  v.Satisfaction, v.Importance, and v.Approval are binary. Satisfaction value 1 reads "satisfied", and 0 "not satisfied". Importance value 1 reads "mandatory", 0 "not mandatory". Approval value 1 reads "approved" and 0 reads "not approved". Each of the $\mathsf{q} = 1, \ldots, n$ Value Types v.q is the set of positive reals.

- Value Assignments are ternary relations over Fragments or relation instances, Value Types, and values of Value Types:

$$\mathbf{V} \subseteq (\mathbf{F} \cup \mathbf{R}) \times \mathbf{T} \times \bigcup_{\mathsf{v.t} \in \mathbf{T}} \mathsf{v.t}.$$

and has the following three partitions

$$(\mathbf{F} \cup \mathbf{R}) \times \{\text{v.Satisfaction}\} \times \{1,0\},$$
$$\mathbf{F} \times \{\text{v.Importance}\} \times \{1,0\}, \text{ and}$$
$$\mathbf{F} \times \bigcup_{\text{q}=1}^{n>1} \text{v.q} \times \mathbb{R}^{+}.$$

Satisfaction values can be assigned to Fragments and relation instances, while values of all other Value Types can be assigned only to Fragments.

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$A ::= x \mid y \mid z \mid \ldots$$
$$B ::= r(A) \mid k(A) \mid s(A)$$
$$C ::= B \text{ influences+ } B$$
$$D ::= B \text{ influences- } B$$
$$G ::= \langle A, E, F \rangle$$

$$H ::= G \text{ alternativeTo } G$$
$$I ::= G \text{ Pref.} J \; G$$
$$J ::= c_1 \mid c_2 \mid \ldots$$
$$K ::= G \text{ CondFor } I$$
$$Z ::= B \mid C \mid D \mid G \mid H \mid I \mid K$$

Mapping

Symbols map to domain elements as follows:

- A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$.

- B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s.
- C and D symbols denote, respectively, positive and negative influence relations.
- E symbols denote Value Types, $\mathscr{D}(E) \in \mathbf{T}$.
- F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction.
- G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$.
- H symbols denote Alternatives, $\mathscr{D}(H) \in$ vr.alt.b.
- I symbols denote instances of preference relations, one per Criterion c,

$$\mathscr{D}(I) \in \bigcup_{\text{crit.c} \in \mathbf{C}} \text{vr.pref.c.}$$

- J symbols denote Criteria, $\mathscr{D}(J) \in \mathbf{C}$.
- K symbols denote conditional preference relation instances, $\mathscr{D}(K) \in$ vr.pref.cond.

Language Services

Those of relations and functions in the language.

Given a model M in L.Elnath, I need the following tuple from it:

$$(\mathbf{Vars}(M), \mathbf{V}, \text{vr.pref}, \text{vr.pref.cond})$$

where $\mathbf{Vars}(M)$ is the set of all variables in the model, $\mathbf{V}$ is a set of Value Assignments, and

$$\text{vr.pref.c} \subseteq \mathbf{V} \times \mathbf{V},$$
$$\text{vr.pref.cond} \subseteq \mathbf{V} \times \bigcup_{\text{crit.c} \in \mathbf{C}} \text{vr.pref.c.}$$

The following function takes the tuple above, and makes a CP-Net from it.

Function: make.CPNet

Make a CPNet f.make.CPNet

Input

(**Vars**(M), **V**, vr.pref, vr.pref.cond).

Do

1. Let **G**(M) be a graph, in which **Vars**(M) is the set of nodes, and every node $x.t \in$ **Vars**(M) is annotated with a so-called Conditional Preference Table (CPT), denoted $CPT(x.\text{v.t})$, where x.v.t is the variable from **Vars**(M).
2. For every variable $x.\text{v.t} \in$ **Vars**(M), find all 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 $x.\text{v.t} \in$ **Vars**(M), define its $CPT(x.\text{v.t})$ by adding every 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 $x.\text{v.t} \in$ **Vars**(M), if its $CPT(x.\text{v.t})$ is not complete, then elicit or otherwise find the missing preferences and vr.pref.cond instances, and add them to $CPT(x.\text{v.t})$.

Output

The CP-Net **G**(M).

Language Services

- s.BestOutcome: The best Outcome is the Outcome returned by an outcome optimisation query [18] on the CP-Net $\mathbf{G}(M)$.
- s.BetterOutcome: The better Outcome is the one returned by a dominance query [18] on the CP-Net $\mathbf{G}(M)$.

Figure 14.3 shows a model in L.Elnath. There are no Value Assignments in the Figure. Each Fragment in the Figure is annotated with two preference relation 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 preferences in the Figure, what is the best Complete 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 Assignmentto the Fragments whose satisfaction values involve conditional preferences. The resulting CP-Net is shown in Figure 14.4. Next, running an outcome optimisation query on the CP-Net in Figure 14.4 will result in the graph in Figure 14.5, 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 14.5 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 14.6.

Figure 14.7 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 14.8, where red circles highlight differences from the satisfaction values in Figure 14.7. 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.

14.6 Summary on Preferences

I introduced binary 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 book:

- How to represent that preferences are conflicting, which is that improvement over one leads to a decrease over another?
- 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?

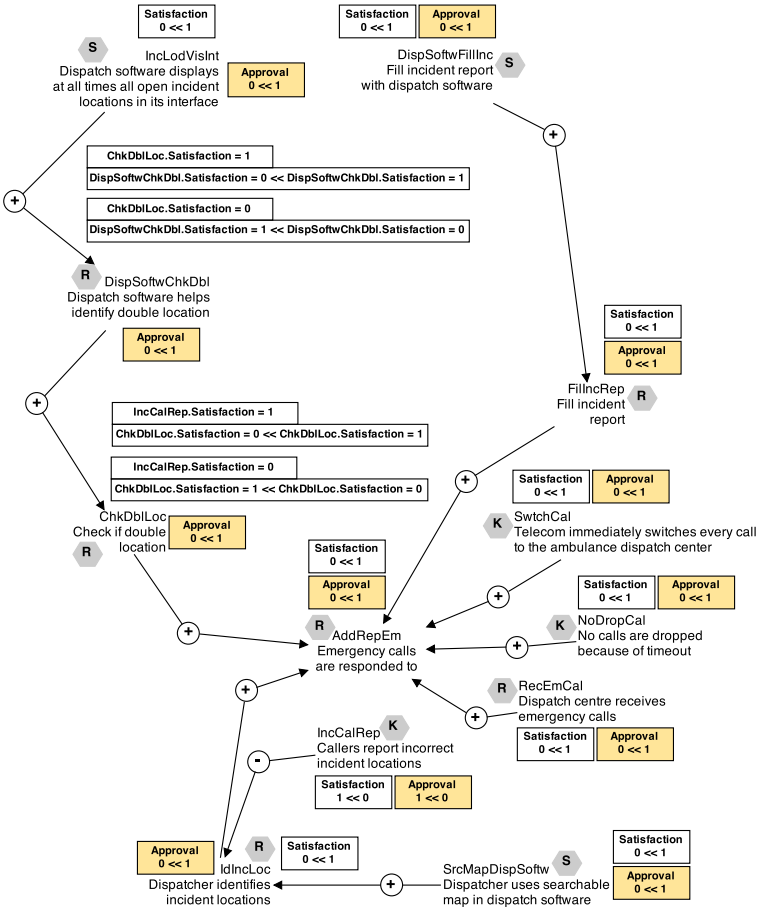


Figure 14.3: Model in L.Elnath, with no Value Assignments.

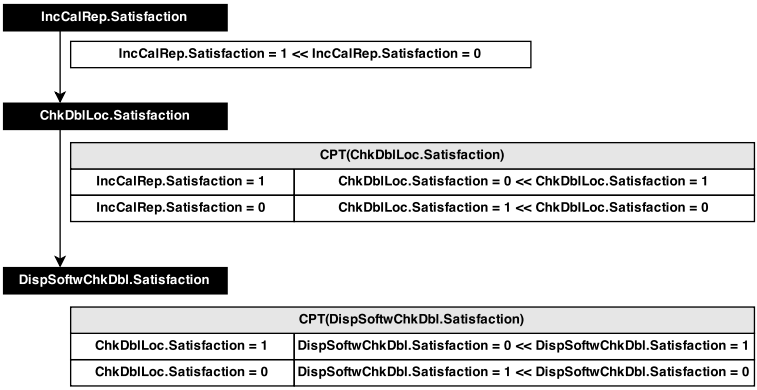


Figure 14.4: CP-Net made from conditional preferences in Figure 14.3.

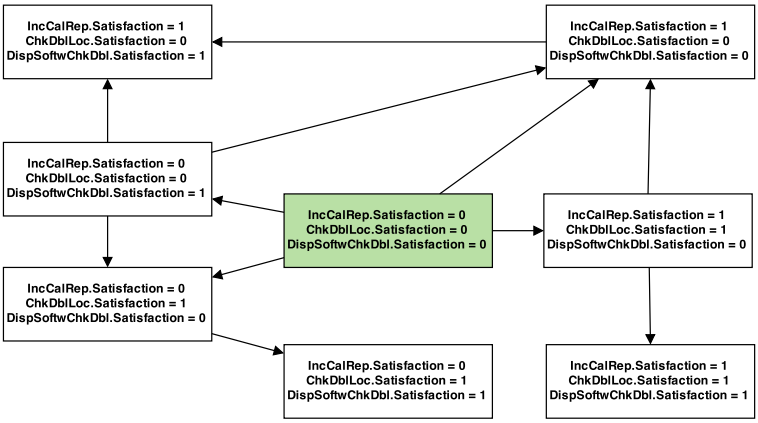


Figure 14.5: Preference graph from the CP-Net in Figure 14.4.

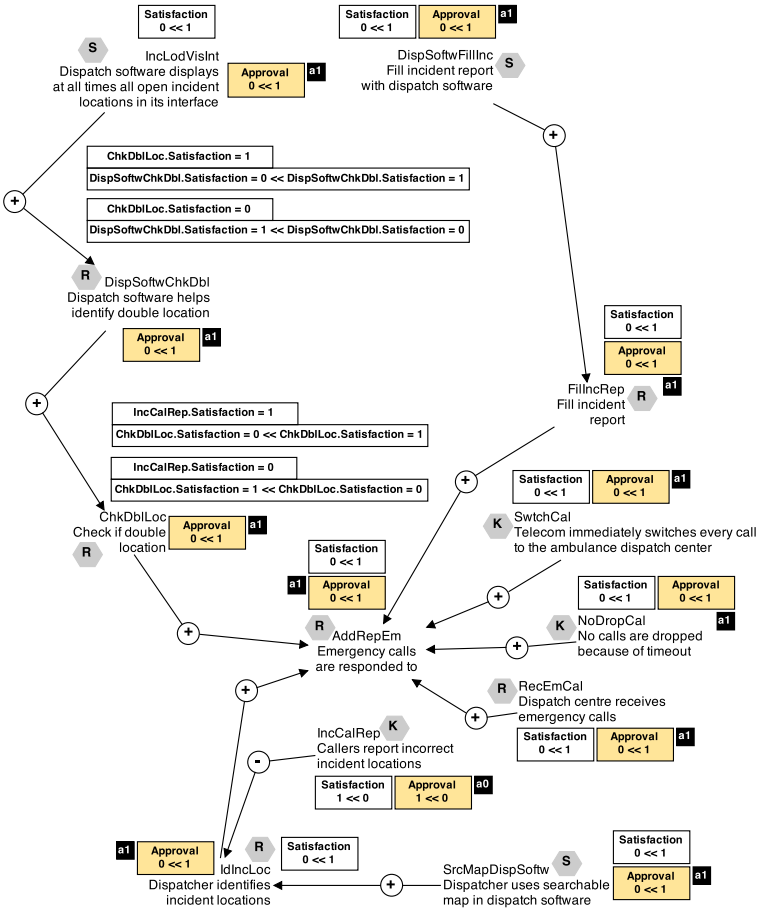


Figure 14.6: Best approval Outcome, assuming 1 is the preferred approval value on all relation instances.

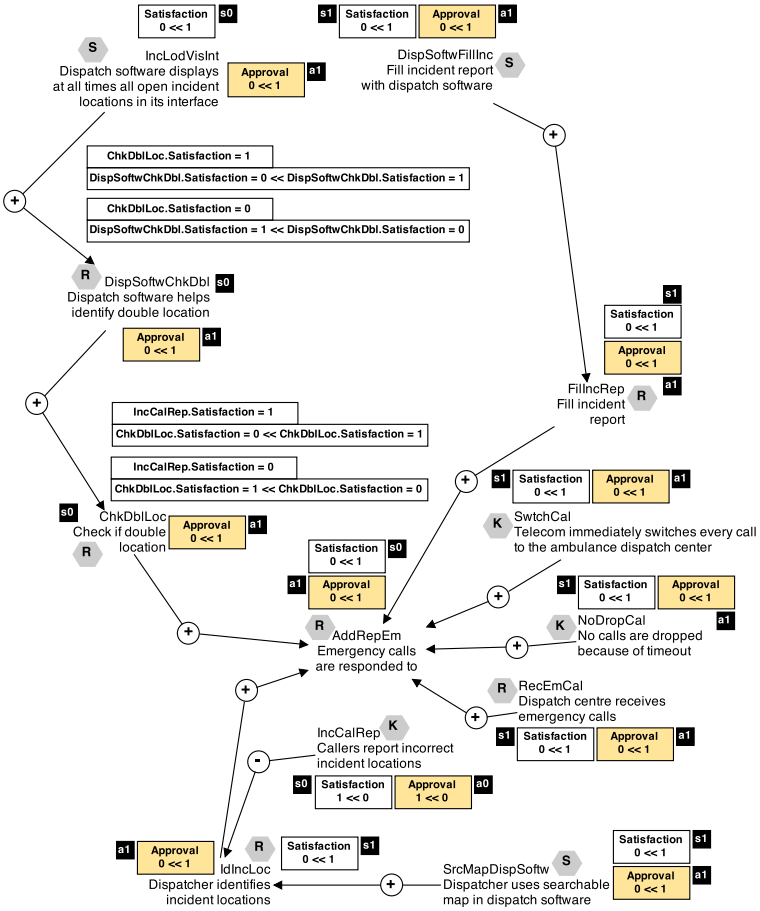


Figure 14.7: Best Outcome according to conditional preferences.

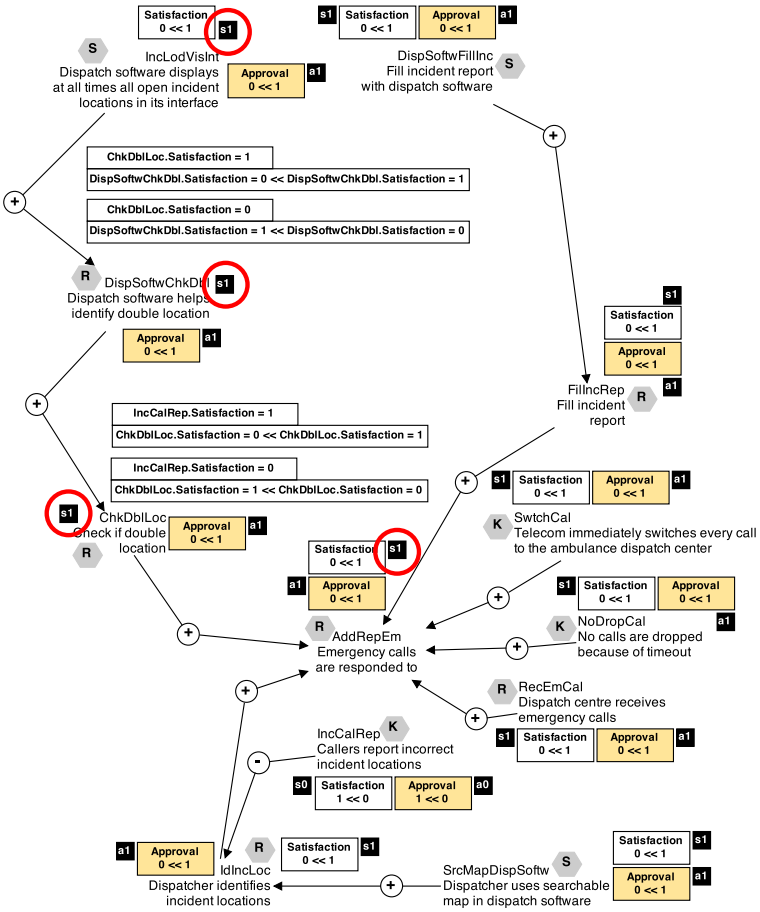


Figure 14.8: Best Outcome which ignores conditional preferences.

Chapter 15

Links to Formal Logic

This Chapter is on how to relate languages in this book to formal logics. The convention in the Chapter 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 Requirements Modelling Languages and formal logics are a recurrent topic in Requirements Engineering. 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 Requirements Modelling Language, map (parts of) it to a theory in some formal logic, in order to answer questions which your Requirements Modelling Language could not. I will look at two among many topics on the relationships between Requirements Modelling Languages 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 15.2)*
2. *How to map a model to CPL theory, if every Fragment maps to a conjunction of formulas of classical propositional logic? (Section 15.3)*
3. *What can be the risks of mapping models to theories? (Section 15.4)*

15.1 Motivation

The overall aim of mapping models to theories is to deliver Language Services which the Requirements Modelling Language 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

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 15.3.

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 [124].[1]

15.2 Models to Theories, Approach One

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.

Consider 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.

[1] Since I want to have a language that represents instances of the DRP, and not some other class of Problems, 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 [155] does not say. I take classical propositional logic to be a conservative choice.

Function: map.infl.impl

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:

$$\begin{aligned} p_1 \wedge \ldots \wedge p_n &\rightarrow x, \\ q_1 \wedge \ldots \wedge q_m \wedge x &\rightarrow \bot. \end{aligned}$$

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. s.WhCPLTh

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 Problem, and then being able to check if a submodel 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.

Submodel

Language: Sirius

Language Modules

L.Sirius

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, f.map.infl.impl

Domain

There is a set of Fragments **F**, a singleton for Value Types

$$\mathbf{T} = \{\text{v.Satisfaction}\},$$

and a set of Value Assignments **V**. Fragments have three partitions, namely requirements, domain knowledge, and specification Fragments, $\mathbf{F} = \text{c.r} \cup \text{c.k} \cup \text{c.s}$ and $\text{c.r} \cap \text{c.k} \cap \text{c.s} = \emptyset$. Influences are over Fragments, $\text{r.inf.pos} \subseteq \mathbf{F} \times \mathbf{F}$, $\text{r.inf.neg} \subseteq \mathbf{F} \times \mathbf{F}$. Value assignments are over Fragments or relation instances, involve a Value Type, and a value, so that

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

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

Syntax

A model M in the language is a set of symbols $M = \{Z_1, \ldots, Z_n\}$, where every ϕ is generated according to the following BNF rules:

$$
\begin{array}{lcl}
A & ::= & x \mid y \mid z \mid \ldots \\
B & ::= & r(A) \mid k(A) \mid s(A) \\
C & ::= & B \text{ influences+ } B \\
D & ::= & B \text{ influences- } B \\
G & ::= & \langle A, E, F \rangle \\
Z & ::= & B \mid C \mid D \mid G
\end{array}
$$

Mapping

A symbols denote Fragments, $\mathscr{D}(A) \in \mathbf{F}$. B symbols are used to distinguish requirements, domain knowledge, and specification Fragments, so that $\mathscr{D}(r(\alpha)) \in$ c.r, $\mathscr{D}(k(\alpha)) \in$ c.k, $\mathscr{D}(s(\alpha)) \in$ c.s. C and D symbols denote, respectively, positive and negative influence relations. E symbols denote Value Types, $\mathscr{D}(E) \in \mathbf{T}$. F symbols denote a value of a Value Type, and as there is one Value Type, $\mathscr{D}(F) \in$ v.Satisfaction. G symbols denote Value Assignments, $\mathscr{D}(G) \in \mathbf{V}$.

Language Services

Those of relations and functions in the language, and

- s.DRPSol: Yes, S is the solution to the Default Problem instance defined by the sub model P, if the following conditions are satisfied:

  1. P and S are submodels of M,
  2. If $\mathbf{F}_{\text{c.k}}$ is the set of atomic CPL propositons, one per domain knowledge Fragment in M, $\mathbf{F}_{\text{c.s}}$ the set of atomic CPL propositons, one per specification Fragment in M, $\mathbf{F}_{\text{c.r}}$ 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) $\mathbf{F}_{c.k}, \mathbf{F}_{c.s}, \Delta \vdash \mathbf{F}_{c.r}$, that is, the Provability Condition is satisfied,

(b) $\mathbf{F}_{c.k}, \mathbf{F}_{c.s}, \Delta \nvdash \bot$, that is, the Consistency Condition is satisfied,

3. S includes all Fragments denoted by the atomic propositions in $\mathbf{F}_{c.s}$, and

4. P includes all Fragments denoted by the atomic propositions in $\mathbf{F}_{c.k} \cup \mathbf{F}_{c.r}$.

Finding an Default Problem 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 Problem 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 $\mathbf{F}_{c.k}$ be the set of atomic CPL propositons, one per domain knowledge Fragment in M, $\mathbf{F}_{c.s}$ the set of atomic CPL propositons, one per specification Fragment in M, $\mathbf{F}_{c.r}$ 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

$$\mathbf{F}_{c.k}, \mathbf{F}_{c.s}, \Delta \vdash \mathbf{F}_{c.r}.$$

The Consistency Condition becomes

$$\mathbf{F}_{c.k}, \mathbf{F}_{c.s}, \Delta \nvdash \bot.$$

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 Problem 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 $K, S, \Delta \nvdash \bot$.

Suppose that you are given the following CPL formulas:

$$
\begin{aligned}
\mathbf{F}_{\text{c.k}} &= \{ \ \mathsf{SwtchCal}, \mathsf{NoDropCal} \ \} \\
\mathbf{F}_{\text{c.s}} &= \{ \ \mathsf{SrcMap}, \mathsf{IncLocVisSw}, \mathsf{UpdOpIncLoc}, \mathsf{FilSwIncRep} \ \} \\
\mathbf{F}_{\text{c.r}} &= \{ \ \mathsf{AddRepEm} \ \} \\
\Delta &= \{ \ \mathsf{FilIncRep} \wedge \mathsf{ChkDblLoc} \wedge \mathsf{IdIncLoc} \wedge \mathsf{RecEmCal} \\
&\quad \wedge \mathsf{NoDropCal} \wedge \mathsf{SwtchCal} \rightarrow \mathsf{AddRepEm}, \\
&\quad \mathsf{SwtchCal} \wedge \mathsf{NoDropCal} \rightarrow \mathsf{RecEmCal}, \\
&\quad \mathsf{SrcMap} \rightarrow \mathsf{IdIncLoc}, \\
&\quad \mathsf{UpdOpIncLoc} \wedge \mathsf{IncLocVisSw} \rightarrow \mathsf{SwIdDuplCal}, \\
&\quad \mathsf{SwIdDuplCal} \rightarrow \mathsf{ChkDblLoc}, \\
&\quad \mathsf{FilISwIncRep} \rightarrow \mathsf{FilIIncRep} \ \}.
\end{aligned}
$$

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 $\mathbf{F}_{\text{c.k}}$, $\mathbf{F}_{\text{c.s}}$, and $\mathbf{F}_{\text{c.r}}$ above, and has influences which mapped to those in Δ above, via f.map.inf.impl? Is there a Default Problem problem instance in M? Is S given above a solution to that Default Problem instance in M? I leave this to you.

15.3 Models to Theories, Approach Two

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 15.2. 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 15.2, since that output amounts to a set of propositional Horn clauses [43].

15.4 Risks of Mapping to Formal 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 15.2.

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 Requirements Modelling Language, 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.

15.5 Summary on Formal Theories

This section briefly mentioned several topics on how models in Requirements Modelling Languages 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?

Bibliography

[1] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 2005.

[2] 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.

[3] Anonymous. Report of the Inquiry Into The London Ambulance Service. Technical report, The Communications Directorate, South West Thames Regional Authority, 1993.

[4] Chimay Anumba, John M Kamara, and Anne-Francoise Cutting-Decelle. *Concurrent engineering in construction projects*. Routledge, 2006.

[5] Kenneth J Arrow, Amartya Sen, and Kotaro Suzumura. *Handbook of Social Choice & Welfare*, volume 2. Elsevier, 2010.

[6] 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.

[7] Jorgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2002.

[8] H Bekić, Dines Bjørner, Wolfgang Henhapl, Cliff B Jones, and Peter Lucas. A formal definition of a pl/i subset. In *Programming Languages and Their Definition*, pages 107–155. Springer, 1984.

[9] Trevor JM Bench-Capon and Paul E Dunne. Argumentation in artificial intelligence. *Artificial intelligence*, 171(10):619–641, 2007.

[10] William L Benoit and Dale Hample. *Readings in argumentation*, volume 11. Walter de Gruyter, 1992.

[11] Patrik Berander and Anneliese Andrews. Requirements prioritization. In *Engineering and managing software requirements*, pages 69–94. Springer, 2005.

[12] 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.

[13] 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.

[14] Barry W Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.

[15] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[16] 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.

[17] Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, 2000.

[18] 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.

[19] 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.

[20] 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.

[21] Russel E Caflisch. Monte carlo and quasi-monte carlo methods. *Acta numerica*, 7:1–49, 1998.

[22] Colin Camerer and Martin Weber. Recent developments in modeling preferences: Uncertainty and ambiguity. *Journal of risk and uncertainty*, 5(4):325–370, 1992.

[23] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information systems*, 27(6):365–389, 2002.

[24] Eugene Charniak. Bayesian networks without tears. *AI magazine*, 12(4):50, 1991.

[25] Li Chen and Pearl Pu. Survey of preference elicitation methods. In *Ecole Politechnique Federale de Lausanne (EPFL), IC/2004/67*, 2004.

[26] 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.

[27] Carlos Iván Chesñevar, Ana Gabriela Maguitman, and Ronald Prescott Loui. Logical models of argument. *ACM Computing Surveys (CSUR)*, 32(4):337–383, 2000.

[28] 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.

[29] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Richard Cox. Representation construction, externalised cognition and individual differences. *Learning and instruction*, 9(4):343–363, 1999.

[35] 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.

[36] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.

[37] Robert Darimont and Axel Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *ACM SIGSOFT Software Engineering Notes*, 21(6):179–190, 1996.

[38] 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.

[39] 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.

[40] 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.

[41] Edsger W Dijkstra. Chapter i: Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press Ltd., 1972.

[42] Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in ai: An overview. *Artificial Intelligence*, 175(7):1037–1052, 2011.

[43] 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.

[44] Didier Dubois and Henri Prade. Possibility theory as a basis for qualitative decision theory. In *IJCAI*, volume 95, pages 1924–1930, 1995.

[45] 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.

[46] SK Eric. *Social modeling for requirements engineering*. Mit Press, 2011.

[47] Neil A Ernst, Alexander Borgida, Ivan J Jureta, and John Mylopoulos. Agile requirements engineering via paraconsistent reasoning. *Information Systems*, 2013.

[48] José Figueira, Salvatore Greco, and Matthias Ehrgott. *Multiple criteria decision analysis: state of the art surveys*, volume 78. Springer, 2005.

[49] 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.

[50] Janos C Fodor and MR Roubens. *Fuzzy preference modelling and multicriteria decision support*, volume 14. Springer, 1994.

[51] 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.

[52] Dedre Gentner and Susan Goldin-Meadow. *Language in mind: Advances in the study of language and thought*. MIT Press, 2003.

[53] Jonathan Ginzburg. Interrogatives: Questions, facts and dialogue. *The handbook of contemporary semantic theory. Blackwell, Oxford*, 1996.

[54] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. In *Conceptual ModelingâĂŤER 2002*, pages 167–181. Springer, 2003.

[55] Lila Gleitman and Anna Papafragou. Language and thought. *Cambridge handbook of thinking and reasoning*, pages 633–661, 2005.

[56] 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.

[57] 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.

[58] Salvatore Greco, Benedetto Matarazzo, and Roman Slowinski. Rough sets theory for multicriteria decision analysis. *European journal of operational research*, 129(1):1–47, 2001.

[59] 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.

[60] Sol J. Greenspan, Alexander Borgida, and John Mylopoulos. A requirements modeling language and its logic. *Inf. Syst.*, 11(1):9–23, 1986.

[61] 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.

[62] Nicola Guarino. Formal ontology, conceptual analysis and knowledge representation. *International journal of human-computer studies*, 43(5):625–640, 1995.

[63] John J Gumperz and Stephen C Levinson. *Rethinking linguistic relativity*. Cambridge University Press, 1996.

[64] 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.

[65] John V Guttag, James J Horning, Stephen J Garland, Kevin D Jones, Andres Modet, and Jeannette M Wing. Larch: languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Citeseer, 1993.

[66] Sven Ove Hansson. Preference logic. In *Handbook of philosophical logic*, pages 319–393. Springer, 2002.

[67] Sven Ove Hansson and Till Grüne-Yanoff. Preferences. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2012 edition, 2012.

[68] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of" semantics"? *Computer*, 37(10):64–72, 2004.

[69] 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.

[70] 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.

[71] Ann M Hickey and Alan M Davis. A unified model of requirements elicitation. *Journal of Management Information Systems*, 20(4):65–84, 2004.

[72] David Hitchcock. Informal logic and the concept of argument. *Philosophy of logic*, 5:101–129, 2006.

[73] C Hoare. Proof of correctness of data representations. *Language Hierarchies and Interfaces*, pages 183–193, 1976.

[74] John E Hopcroft and Robert E Tarjan. Efficient algorithms for graph manipulation. 1971.

[75] Anthony Hunter. Paraconsistent logics. In *Reasoning with Actual and Potential Contradictions*, pages 11–36. Springer, 1998.

[76] 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.

[77] United Kingdom Hydrograph, United Kingdom Hydrographic Office, and U S Naval Observatory. *2010 Nautical Almanac: Commercial Edition*. Paradise Cay Publications, 2009.

[78] Kenneth E Iverson. Notation as a tool of thought. *ACM SIGAPL APL Quote Quad*, 35(1-2):2–31, 2007.

[79] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[80] David Jonassen. Using cognitive tools to represent problems. *Journal of research on Technology in Education*, 35(3):362–381, 2003.

[81] David Jonassen, Johannes Strobel, and Chwee Beng Lee. Everyday problem solving in engineering: Lessons for engineering educators. *Journal of engineering education*, 95(2):139–151, 2006.

[82] 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.

[83] 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.

[84] Ivan J Jureta, Alexander Borgida, Neil A Ernst, and John Mylopoulos. The requirements problem for adaptive systems. *ACM Transactions on Management Information Systems (TMIS)*, 5(3):17, 2014.

[85] 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.

[86] Ivan J Jureta, Stéphane Faulkner, and Pierre-Yves Schobbens. A more expressive softgoal conceptualization for quality requirements analysis. In *Conceptual Modeling-ER 2006*, pages 281–295. Springer, 2006.

[87] 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.

[88] 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.

[89] Daniel Kahneman and Amos Tversky. Prospect theory: An analysis of decision under risk. *Econometrica: Journal of the Econometric Society*, pages 263–291, 1979.

[90] Nikos Karacapilidis and Dimitris Papadias. Computer supported argumentation and collaborative decision making: the hermes system. *Information systems*, 26(4):259–277, 2001.

[91] Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, 39(14):939–947, 1998.

[92] Paul Kay and Willett Kempton. What is the Sapir-Whorf hypothesis? *American Anthropologist*, 86(1):65–79, 1984.

[93] 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.

[94] 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.

[95] Bryan Lawson. *How designers think: the design process demystified*. Routledge, 2006.

[96] 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.

[97] Jintae Lee and Kum-Yew Lai. What's in design rationale? *Human–Computer Interaction*, 6(3-4):251–280, 1991.

[98] 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.

[99] 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.

[100] Sotirios Liaskos, Sheila A McIlraith, 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.

[101] Sarah Lichtenstein and Paul Slovic. *The construction of preference*. Cambridge University Press, 2006.

[102] Panagiotis Louridas and Pericles Loucopoulos. A generic model for reflective design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(2):199–237, 2000.

[103] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[104] Matthew McGrath. Propositions. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[105] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In *Argumentation in artificial intelligence*, pages 105–129. Springer, 2009.

[106] Dennis C Mueller. *Public choice: an introduction*. Springer, 2004.

[107] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):325–362, 1990.

[108] 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.

[109] John Neter, William Wasserman, and Michael H Kutner. Applied linear regression models. 1989.

[110] 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.

[111] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[112] Eric Pederson, Eve Danziger, David Wilkins, Stephen Levinson, Sotaro Kita, and Gunter Senft. Semantic typology and spatial conceptualization. *Language*, pages 557–589, 1998.

[113] 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.

[114] John L Pollock. Defeasible reasoning. *Cognitive science*, 11(4):481–518, 1987.

[115] Frederic Portoraro. Automated reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2014 edition, 2014.

[116] Henry Prakken and Gerard Vreeswijk. Logics for defeasible argumentation. In *Handbook of philosophical logic*, pages 219–318. Springer, 2002.

[117] Balasubramaniam Ramesh and Vasant Dhar. Supporting systems development by capturing deliberations during requirements engineering. *Software Engineering, IEEE Transactions on*, 18(6):498–510, 1992.

[118] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, 2001.

[119] Nicholas Rescher. The logic of preference. In *Topics in Philosophical Logic*, pages 287–320. Springer, 1968.

[120] 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.

[121] Horst WJ Rittel and Melvin M Webber. Dilemmas in a general theory of planning. *Policy sciences*, 4(2):155–169, 1973.

[122] William N Robinson, Suzanne D Pawlowski, and Vecheslav Volkov. Requirements interaction management. *ACM Computing Surveys (CSUR)*, 35(2):132–190, 2003.

[123] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

[124] Stewart Shapiro. Classical logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2013 edition, 2013.

[125] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[126] S Buckingham Shum. Design argumentation as design rationale. *The encyclopedia of computer science and technology*, 35(20):95–128, 1996.

[127] 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.

[128] Guillermo Simari and Iyad Rahwan. Argumentation in artificial intelligence. 2009.

[129] Guillermo R Simari and Ronald P Loui. A mathematical treatment of defeasible reasoning and its implementation. *Artificial intelligence*, 53(2):125–157, 1992.

[130] Herbert A Simon. The structure of ill-structured problems. In *Models of discovery*, pages 304–325. Springer, 1977.

[131] Guttorm Sindre and Andreas L Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[132] 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.

[133] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[134] Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer, 2010.

[135] Mark Staples. Critical rationalism and engineering: ontology. *Synthese*, pages 1–25, 2014.

[136] 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.

[137] 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.

[138] Barbara G Tabachnick, Linda S Fidell, et al. Using multivariate statistics. 2001.

[139] Alfred Tarski. The semantic conception of truth: and the foundations of semantics. *Philosophy and phenomenological research*, 4(3):341–376, 1944.

[140] Richard Thaler. Toward a positive theory of consumer choice. *Journal of Economic Behavior & Organization*, 1(1):39–60, 1980.

[141] Amos Tversky and Daniel Kahneman. Judgment under uncertainty: Heuristics and biases. *science*, 185(4157):1124–1131, 1974.

[142] Amos Tversky, Paul Slovic, and Daniel Kahneman. The causes of preference reversal. *The American Economic Review*, pages 204–217, 1990.

[143] JFAK van Benthem and Alice Ter Meulen. *Handbook of logic and language*. Elsevier, 1996.

[144] 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.

[145] 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.

[146] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, 2000.

[147] Douglas N Walton. *Informal logic: A handbook for critical argumentation*. Cambridge University Press, 1989.

[148] Philippe Weil. Nonexpected utility in macroeconomics. *The Quarterly Journal of Economics*, pages 29–42, 1990.

[149] 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.

[150] Jeannette M Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[151] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[152] Eric Siu-Kwong Yu. *Modelling strategic relationships for process reengineering*. PhD thesis, University of Toronto, 1995.

[153] Eric SK Yu. Modeling organizations for information systems requirements engineering. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 34–41. IEEE, 1993.

[154] 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.

[155] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

[156] Milan Zeleny and James L Cochrane. *Multiple criteria decision making*, volume 25. McGraw-Hill New York, 1982.

[157] Jiajie Zhang. The nature of external representations in problem solving. *Cognitive science*, 21(2):179–217, 1997.