Techne Framework for the Declarative Specification of Modelling Languages for Requirements Engineering

Ivan J. Jureta[*†] Alexander Borgida[‡] Neil A. Ernst[§] John Mylopoulos[¶]

July 17, 2013

Abstract

Requirements Engineering (RE) focuses on eliciting, modelling and analyzing the purpose of a system-to-be in order to derive its specification. RE uses representations of requirements, that is, requirements models. These models are made using Requirements Modeling Languages (RMLs). RMLs proposed by now have been specified in different ways, which made it difficult to relate, compare, change, and merge them. This paper proposes the Techne Framework (TF) for the formal specification of RMLs. The paper shows how to use TF (i) to specify a new RML, (ii) to specify an existing RML, (iii) to specify changes to RMLs, (iv) to specify merged RMLs, and (v) to analyze RMLs by analyzing interesting properties of their TF specifications.

[*]Fonds de la Recherche Scientifique – FNRS, Brussels.
[†]Department of Business Administration, University of Namur; ivan.jureta@unamur.be.
[‡]Department of Computer Science, Rutgers University; borgida@cs.rutgers.edu
[§]Software Engineering Institute, Carnegie Mellon; nernst@cs.ubc.ca
[¶]Department of Information Engineering and Computer Science, University of Trento; jm@disi.unitn.it

Contents

1 Introduction

Requirements Engineering (RE) research focuses on issues that arise when it is necessary to elicit, model, and analyze the purpose of a system-to-be in order to derive its specification. Such issues are present both when engineering new systems, and when changing existing ones.

A basic result in RE research is knowledge on how to solve these issues. Surveys show that there is a considerable amount of such knowledge [41, 10] in response to the various issues of interest to the field [51], including, for example: "How to gather information about a system-to-be and its environment?" [21, 27, 15], "Which of the gathered information is relevant to RE?" [13, 52, 31], "How to clarify the elicited information, to avoid such problems as vagueness and ambiguity?" [40, 37, 30], "How to determine which requirements have highest priority, for whom, and why?" [34, 2, 26], "How to help stakeholders agree on common priorities over requirements?" [36, 4, 32], "How to distribute the responsibility for the satisfaction of requirements to the system-to-be, systems it might interact with, and people in its environment?" [13, 9, 19], "How to estimate costs, risks, and deadlines for making systems that satisfy requirements?" [6, 5, 43], "How to evaluate how complete requirements are, and if any important requirements may have been missed?" [24, 44, 53], "How to evaluate if the requirements are consistent, and to manage inconsistent requirements?" [18, 25, 45], "How to specify and compare alternative strategies to satisfy the same requirements?" [40, 37, 39], "How to evaluate the quality to which requirements would be satisfied by a system-to-be?" [7, 40, 35], "How to check if a system-to-be specification satisfies requirements?" [20, 19, 17], "How to keep track of changes to requirements, reasons for these changes, their impact on existing requirements and systems, and how to propagate these changes in an existing specification of a system-to-be?" [22, 42, 12], "How to do RE for systems capable of adapting to their enviroment?" [11, 46, 8], and others.

It is usually necessary to have a representation of requirements, that is, a *requirements model* when solving these issues. When requirements are elicited, they are added to such models; when they are negotiated, the parties involved use the model to facilitate communication; the model is a basis for estimating costs, risks, and deadlines; it is used to evaluate completeness and clarity; and so on.

Requirements Modeling Languages (RMLs) are used to make requirements models. Research on RMLs goes back to the original framework for requirements models, RMF [23]. Many different RMLs have been proposed since, including ERAE [16], KAOS [13], *i** [48], LQCL [28], and Techne [29].

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

This diversity highlights two key issues for the research on the design of RMLs. Firstly, it is unlikely that there is one best RML, which we have somehow failed to discover by now; rather, different RMLs may be useful for different domains, system classes, projects, organizations, etc. There is therefore a need to make potentially many RMLs on demand[1], and so, a need for more general knowledge about how to make new RMLs. Secondly, once we accept that a proliferation of RMLs is not undesirable, it becomes critical to develop a body of knowledge on how to relate, extend, compare, and analyze RMLs in a systematic way.

This paper proposes a framework, called the Techne Framework (TF), for the specification of RMLs. The motivation for producing such specifications is to facilitate the presentation, extension, merging, and comparison of RMLs, as well as deal with ambiguity, imprecision, incompleteness, and inconsistency in them.

[1] In this paper alone, we specify 12 RMLs: **T1**, **iS1**, **iS2**, **iS3**, **T1t**, **T1qc**, **T1op**, **T1r**, **T1me**, **T0**, **TS1**, **T1rx**.

TF is used for the *declarative* specification of RMLs. This means that a specification will give (i) necessary conditions for the RML to apply to a problem encountered during RE, (ii) the properties of the solution sought when using that RML, and (iii) the rules to satisfy when using the RML. The specification will *not* describe steps to take to find the solution, but allow different paths to solutions. Which of these paths are better than others is a question for a *procedural* specification of RE methods (also called RE techniques [47]) that use these RMLs, and is not covered in this paper.

TF is based on the idea that an RML embeds three kinds of knowledge for solving problems in RE; we call them Problem-Solving, Interface, and Domain knowledge. *Problem-Solving Knowledge* is the general knowledge we apply whenever we identify an instance of a problem we learned to solve before. *Domain knowledge* is what we know about the concrete situation in which we observed the problem instance. *Interface knowledge* tells us how to combine Problem-Solving and Domain knowledge in order to solve the problem instance.

For example, the specifics we know about the particular systems engineering project are our Domain Knowledge, including, for example, stakeholders, their expectations from the system-to-be, regulations applicable to the system-to-be, business processes it may need to support, etc. If we want to use i^* for that project, then all we know about i^* is our Problem-Solving Knowledge, including that we should look for for agents, and find out how they depend on one another to achieve goals, execute tasks, and deliver resources. i^* will not be able to capture all we know about the specific project: it is Interface Knowledge that tells us which of the available Domain Knowledge we should use when making an i^* model of the system-to-be.

A TF specification of an RML has three parts, corresponding to the three knowledge types:

1. The Domain part includes (i) the Knowledge Base (KB) that holds the representation of Domain knowledge, (ii) the language for writing that KB, and (iii) the inference rules to draw conclusions from the KB. The contents of the KB changes from one problem instance to the next, while the representation language and inference rules remain the same.

2. The Interface part includes functions, called *selectors*, which return statements from the Domain KB. They select only those statements manipulated with Problem-Solving Knowledge.

3. The Problem-Solving part also has a KB, a predicate language for writing that KB, inference rules, and rules that the KB has to satisfy. The key ideas of the RML are reflected in the selection of predicates allowed in the language, and in the rules that relate the predicates. Terms in predicates are members of the Domain KB: if x is a proposition in the Domain KB, and there is a selector which returns x, then there will be statements over x in the Problem-Solving KB. For example, if x is the propostion "an ambulance should be dispatched to a reported incident location", then there may be a statement in the Problem-Solving KB which says "x is a requirements and the system-to-be must satisfy it", written $\mathsf{Requirement}(x)$.

A TF specification is *formal*, in the sense that all its parts are written using a formalism that has a well-defined mathematical basis. As we will show in the paper, this helps us extend, merge, and compare RMLs. It also helps us define and analyze interesting properties of RMLs via formal properties of their TF specifications, including:

- *Conciseness*, which fails if there are parts of the RML that can be removed, without affecting our ability to solve the problem with what remains.

- *Clarity*, which fails if we do not always know whether we have found a solution. Clarity succeeds if we can determine, at all times when applying the RML, if we have found at least one solution.

- *Decisiveness*, which fails if we can find more than one solution instance for the same problem instance. If so, it means that the RML does not produce a unique solution to a problem.

A TF specification of an RML *neither is, nor is intended to be the unique or universal* specification of it. An RML reflects the knowledge of its designers, and a TF specification thereof as an attempt to

represent some of that knowledge. We accept that there can be different TF specifications of what seems to be the same RML. We leave for future work the evaluation of the *coverage* of some RML by its TF specification.

This paper is organized as follows. Section 2 introduces basic terminology and conventions. Section 3 defines the Techne Framework. Section 4 illustrates the use of TF for the specification of a new RML. Section 5 illustrates the use of TF to specify an existing RML. Section 6 illustrates how to use TF to specifiy changes of RMLs, whereby change we mean adding or removing capabilities to RMLs. Section 7 illustrates how to merge the TF specifications of different RMLs to create a new RML. Section 8 discusses how to analyze RMLs by analyzing the properties of their TF specifications.

2 Terminology and Conventions

All RMLs in this paper are specified with TF. All specifications are declarative and formal, in the sense explained in Section 1.

A TF specification has three parts, **Domain Knowledge** (denoted **D**), **Interface Knowledge** (**I**), and **Problem-Solving Knowledge** (**S**). None of these are **Knowledge Bases** (KBs) themselves, but **D** and **S** include KBs. We discuss the reasons for this in Section 3.

A KB is a formal representation of a set of propositions [38]. **Proposition** refers to the meaning of a declarative natural language sentence. A proposition is represented by an **expression**.

As we can use different formal languages in TF specifications, there are different kinds of expressions. If a language is propositional, such as in propositional logic, we will say that a proposition is represented either by an **atom**, which is a propositional variable, or a **formula**, which combines atoms according to the syntax of the language. In case of a predicate language, such as first-order logic, we will say that a proposition is represented by a **ground formula**, a formula with no free variables. A **free formula** is a formula with free variables.

An expression is either an atom, a formula, or a ground formula, and the context will make it clear which it is; when we write **expressions**, we are referring to a set of expressions.

Language refers to a set of expressions that all satisfy a specific grammar. A **grammar** is a set of rules for generating expressions. A **formalism** is a combination of language and inference rules on its expressions. By **inference rule**, we mean any function which takes one, and returns another (potentially equal) set of expressions. A **consequence relation** relates a set of expressions, called **premises**, to an expression, called **conclusion**, if and only if there is a **proof** from the former to the latter; such a proof is a finite sequence of expressions, each of which is an axiom of the language over which the consequence relation is defined, or follows from the preceding expressions in the sequence by the application of an inference rule.

Given a consequence relation and a set of expressions X, the **closure of** X is a set Ythat includes all expressions which are conclusions of X according to that consequence relation.

We use lowercase Latin letters to refer to expressions and free formulas. We use uppercase Latin letters to denote KBs or sets of free formulas. What exactly we refer to will be clear from the context.

When we write that **an expression** x **is in** Problem-Solving Knowledge, we mean that x is in some set which is part of **S**. Analogous reading applies to expressions which we say are in **D**. It does not apply to **I**, as Interface Knowledge includes only functions.

When we say that **Problem-Solving Knowledge is about Domain Knowledge**, we mean that expressions in the former range over expressions of the latter. For this to be the case, expressions in **S** need to be those of a predicate language, and the set of allowed constants in that language must include all expressions in Domain Knowledge. In other words, at least a part of (if not, in some cases the entire) universe of discourse of Problem-Solving Knowledge needs to be restricted to expressions in **D**, that is, to what we know about the specific problem instance.

For example, suppose that **S** can include expressions that use a unary predicate $\mathsf{Requirement}(\cdot)$, such that $\mathsf{Requirement}(x)$ reads "x is a requirement". Let y be an atom in **D**, and refer to the proposition "when an incident is reported, an ambulance is dispatched", then the expression y is in **D**, but there may also be an expression $\mathsf{Requirement}(y)$ in **S**, through which we convey the idea

that, according to Problem-Solving Knowledge, y is a requirement. Then, there may be a rule in $\mathbf{S}$, according to which if a requirement in a KB X which holds Domain Knowledge expressions is not satisfied, then there is no solution in that KB, which may be written:

$$\forall a \in X \ \mathsf{Requirement}(a) \wedge \neg \mathsf{Satisfied}(a) \Rightarrow \nexists Y \subseteq X \ \mathsf{Solution}(Y).$$

It is by defining predicates that range over Domain expressions, and rules over these predicates, that we capture Problem-Solving Knowledge with a declarative specification.

3 Techne Framework

TF suggests (i) questions that an RML specification needs to answer, and (ii) structure that this specification should have. The structure organizes the answers to the questions.

Table 1 gives the structure of a TF specification. It lists the three parts of the specification, and their respective components. Each component answers questions, as follows:

1. *What do we know about the concrete situation we are in, and which may include a problem instance?* Domain Knowledge answers this question by answering the following, more specific ones:

   (a) *What do we know about that concrete situation?* Answer: Domain KB, denoted X, as it includes a representation of propositions about the situation we are in.

   (b) *How do we represent what we know about the concrete situation?* Answer: Domain language, $\mathcal{L}_{\mathbf{D}}$, as it defines the language we use to write the Domain KB.

   (c) *What can we conclude from what we know about the concrete situation?* Answer: All expressions in the closure of X, and to compute any member of the closure, we need the Domain consequence relation, $\mid\!\sim_{\mathbf{D}}$, defined from inference rules that we allow over Domain Knowledge. Depending on our choice of the Domain formalism, it may be more relevant to use the satisfiability relation, instead of a consequence relation. When we do so, we will denote the latter with $\models\!\!\approx_{\mathbf{D}}$.

2. *Given what we know about the* RML*, is there a problem instance we can solve with that* RML *in the concrete situation we are in?* Answer: Problem concept, $\mathsf{Problem}(\cdot)$, because it defines properties that should be satisfied by the Domain KB in order for a problem instance to exist. If $\mathsf{Problem}(X)$ is true, then there is a problem instance in X.

3. *Which of what we know about the concrete situation do we use to solve the problem instance?* Answer: Selectors in Interface Knowledge, as they will return, from Domain KB or its closure, the expressions useful to solve the problem.

4. *Is there a solution to the problem in the concrete situation we are in?* Answer: Solution concept, $\mathsf{Solution}(\cdot,\cdot)$, as it defines properties that should be satisfied by the Domain KB in order for a solution instance to exist. If $\mathsf{Solution}(Y, X)$ is true, then Y is a solution to the problem instance in X.

5. *How should we structure what we know about the concrete situation when we are solving the problem?* Answer: Sorts and Relations. Sorts gives categories to use for the classification of knowledge we consider – through the selectors – as relevant for the resolution of the problem. Relations gives relations that may exist, or that we may want to establish between the knowledge we are considering while searching for solution instances.

6. *How do we know how close we are to one or more solution instances?* Answer: Statuses, as its members define the properties that fragments of selected Domain KB should have, when there is no solution instance, and when there are parts of one or more solution instances.

Table 1: Structure of a TF specification.

PART	COMPONENT	NAME AND DEFINITION
D		*Domain Knowledge*, includes:
	$\mathcal{L}_{\mathbf{D}}$	*Domain language*: All expressions for the representation of Domain propositions.
	$\mid\!\sim_{\mathbf{D}}$ or $\mid\!\approx_{\mathbf{D}}$	*Domain consequence or satisfiability relation*: Consequence or satisfiability relation defined from inference rules over $\mathcal{L}_{\mathbf{D}}$.
	X	*Domain* KB: KB such that $X \subseteq \mathcal{L}_{\mathbf{D}}$.
I		*Interface Knowledge*: Set of functions. Each is called a *selector* and is from $\mathcal{L}_{\mathbf{D}}$ expressions to $\mathcal{L}_{\mathbf{D}}$ expressions.
S		*Problem-Solving Knowledge*, includes:
	$\mathcal{L}_{\mathbf{S}}$	*Problem-Solving language*: All expressions for the representation of Problem-Solving propositions.
	$\mid\!\sim_{\mathbf{S}}$ or $\mid\!\approx_{\mathbf{S}}$	*Problem-Solving consequence or satisfiability relation*: Consequence or satisfiability relation defined from inference rules over $\mathcal{L}_{\mathbf{S}}$.
	$\langle X \rangle_{\mathbf{S}}$	*Problem-Solving* KB: A KB such that $\langle X \rangle_{\mathbf{S}} \subseteq \mathcal{L}_{\mathbf{S}}$ and all expressions in $\langle X \rangle_{\mathbf{S}}$ are about the Domain Knowledge expressions in X.
	Sorts	*Sorts*: A set of predicates, each referring to a sort.
	Statuses	*Statuses*: A set of predicates, each referring to a status of a sort.
	Relations	*Relations*: A set of predicates, each referring to a relation.
	Evaluations	*Evaluations*: A set of predicates, each referring to an evaluation.
	Problem	*Problem concept*: A predicate, such that if $\mathsf{Problem}(X)$ is true, then there is a problem instance to solve in X.
	Solution	*Solution concept*: A predicate, such that if $\mathsf{Solution}(Y, X)$ is true, then there is a solution instance Y in the problem in X.
	Rules	*Rules*: A set of expressions in $\mathcal{L}_{\mathbf{S}}$ that all Problem-Solving KBs must be consistent with.

7. *If we have more than one solution instance, how can we compare them?* Answer: Evaluations, as they define order relations over solution instances, and allow the aggregation of these orders into a total order over all solution instances.

8. *How do we know if our way of solving the problem instance is correct?* Answer: Rules, as they define the rules that we need to satisfy while solving the problem instance.

To make a TF specification of an RML amounts to answering the questions above, and filling in the answers to their respecitive TF specification components. We do this in Section 4 for a new RML, and in Section 5 for an existing RML.

4 RML Specification

The TF specification in this section is for a new RML that can be used to solve the following problem: Given goals and assumptions about the environment in which the system-to-be will run, find tasks that need to be executed to achieve the goals and maintain the assumptions. The problem is inspired by Zave & Jackson's [52] statement of the so-called requirements problem. The RML is called **T1**.

We start with Domain Knowledge of **T1**. To make the Domain KB easily readable, we take the language of classical propositional logic as the Domain language, and that logic's consequence relation, denoted $\vdash$, as the Domain consequence relation. As we are not looking at a particular problem instance for the moment, the Domain KB is some set of expressions in the Domain language. This results in the following **D** part of **T1**.

$\mathbf{D_{T1}} = (\mathcal{L}_{\text{CPL}}, \vdash, X)$, where:

- $\mathcal{L}_{\text{CPL}}$ is the language of classical propositional logic.
- $\vdash$ is the consequence relation of classical propositional logic.
- X is some subset of $\mathcal{L}_{\text{CPL}}$.

The informal problem statement mentions goals, assumptions, and tasks. We want to be able to say that an atom in the Domain KB is either of these; we need a selector that returns atoms only. To solve the problem, we also need to be able to say that, given some tasks and assumptions, we can conclude that they achieve or not some goals. We need this to answer such questions as "How is a goal r achieved?" That is, we need to be able to talk in Problem-Solving Knowledge about proofs that we can define from the Domain KB. It follows that we need selectors for proofs, and once we have that, it will be useful to select only premises, or only the conclusion of a proof. This leads to the following Interface Knowledge for **T1**.

$\mathbf{I_{T1}} = \{\mathsf{atoms}(\cdot), \mathsf{minproofs}(\cdot), \mathsf{premises}(\cdot,\cdot), \mathsf{conclusion}(\cdot,\cdot)\}$, where:

- $\mathsf{atoms}(Y)$ returns the set of atoms in $Y \subseteq \mathcal{L}_{\text{CPL}}$.
- $\mathsf{minproofs}(Y) = \{(X_1, z_1), \ldots, (X_{n\geq 0}, z_{n\geq 0})\}$, where for every $i = 1, \ldots, n$, (X_i, z_i) satisfies the following conditions:
  1. $X_i \subseteq Y$ and $Y \subseteq \mathcal{L}_{\text{CPL}}$,
  2. z_i is either a member of $\mathcal{L}_{\text{CPL}}$, or $\bot$ (read "inconsistency"),
  3. $X_i \vdash z_i$,

4. there is no $X_i' \subset X_i$ such that $X_i' \vdash z_i$, i.e., X_i includes only the formulas necessary to prove z_i.

In a summary, $\mathsf{minproofs}(Y)$ returns, for a given set Y of Domain expressions, the set of all pairs (X_i, z_i), in which X_i is the minimal set of expressions from Y that are sufficient to deduce z_i using $\vdash$.

- $\mathsf{premises}(X, z) = X$ is the set of premises in a proof of z from X, if there is a proof.
- $\mathsf{conclusion}(X, z) = z$ is the conclusion in a proof of z from X, if there is a proof.

We use classical first-order logic (CFOL hereafter) as the formalism in Problem-Solving Knowledge, so that Problem-Solving language is the language of CFOL, and the Problem-Solving KB a set of expressions in that language. While we are interested in proofs in Domain Knowledge, our concern in Problem-Solving Knowledge is only if the Problem-Solving KB satisfies or fails some properties of interest. We consequently replace the Problem-Solving consequence relation by the satisfiability relation $\models$. This gives us the first components of $\mathbf{S}$ in **T1**, as follows.

$$\begin{aligned}\mathbf{S_{T1}} = (&\mathcal{L}_{\text{CFOL}}, \models, \langle X\rangle_{\mathsf{S}}, \mathsf{Sorts}, \mathsf{Statuses}, \mathsf{Relations}, \mathsf{Evaluations},\\ &\mathsf{Problem}(\cdot), \mathsf{Solution}(\cdot,\cdot), \mathsf{Rules})\end{aligned}$$

where:

- $\mathcal{L}_{\text{CFOL}}$ is the language of CFOL.
- $\models$ is the satisfiability relation of CFOL.
- $\langle X\rangle_{\mathsf{S}}$ is a set of CFOL expressions. All constants in these expressions are members of Domain KB X, or of its closure.

Following the problem statement, we have the goal, assumption, and task sorts. A goal can be achieved, an assumption maintained, and a task executed. This gives three statuses, one for each sort. Only atoms from Domain Knowledge can be sorted, not other expressions. This is because we do not use composite sorts: for example, let $p \wedge q$ be a Domain expression, and let p be a requirement and q an assumption; a composite sort would be the sort of $p \wedge q$.

$$\begin{aligned}\mathsf{Sorts} = \{&\mathsf{G} : \mathsf{atoms}(\mathcal{L}_{\text{CPL}}) \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{A} : \mathsf{atoms}(\mathcal{L}_{\text{CPL}}) \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{T} : \mathsf{atoms}(\mathcal{L}_{\text{CPL}}) \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\}\}\\ \mathsf{Statuses} = \{&\mathsf{Achieved} : \{x \mid \mathsf{G}(x, Y) = \text{True}\} \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{Maintained} : \{x \mid \mathsf{A}(x, Y) = \text{True}\} \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{Executed} : \{x \mid \mathsf{T}(x, Y) = \text{True}\} \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\}\}\end{aligned}$$

where:

- $\mathsf{G}(x, Y) = \text{True}$ reads "x is a Goal in Y", "x is not a Goal in Y" otherwise.

- $\mathsf{A}(x, Y)$ = True reads "x is an Assumption in Y", "x is not an Assumption in Y" otherwise.
- $\mathsf{T}(x, Y)$ = True reads "x is a Task in Y", "x is not a Task in Y" otherwise.
- $\mathsf{Achieved}(x, Y)$ = True reads "Goal x is achieved in Y", "Goal x is not achieved in Y" otherwise.
- $\mathsf{Maintained}(x, Y)$ = True reads "Assumption x is maintained in Y", "Assumption x is not maintained in Y" otherwise.
- $\mathsf{Executed}(x, Y)$ = True reads "Task x is executed in Y", "Task x is not executed in Y" otherwise.

We want all atoms to be sorted, which leads to the following first rule.

$$\mathsf{Rules} = \{\mathsf{Rule[GAT]}\}$$

where:

$$\begin{aligned}\mathsf{Rule[GAT]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ \forall x \in \mathsf{atoms}(Y)\\ (\mathsf{G}(x, Y) \vee \mathsf{A}(x, Y) \vee \mathsf{T}(x, Y)) \wedge \neg(\mathsf{G}(x, Y) \wedge \mathsf{A}(x, Y) \wedge \mathsf{T}(x, Y))\end{aligned}$$

Status of a sorted expression depends on its relations to other sorted expressions. It is useful to think of any of the three statuses as being either positive or negative: achieved, maintained, and executed are positive, while not achieved, not maintained, and not executed are negative. We use two relations, Break and Build. Break is undirected, and when it relates some sorted atoms, then this means it is not possible for all of these atoms to have positive statuses together. Build is directed from one or more sorted atoms to a single other sorted atom. If there is a Build relation from some sorted atoms to another sorted atom, then the latter will get a positive status if the former all have a positive status.

$$\begin{aligned}\mathsf{Relations} = \{&\mathsf{Build} : \wp(\mathcal{L}_{\mathbf{D}}) \times \mathcal{L}_{\mathbf{D}} \times \wp(\mathcal{L}_{\mathbf{D}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{Break} : \wp(\mathcal{L}_{\mathbf{D}}) \times \wp(\mathcal{L}_{\mathbf{D}}) \longrightarrow \{\text{True}, \text{False}\}\}\end{aligned}$$

where:

- $\mathsf{Build}(X, z, Y)$ = True reads "there is a positive relation between X and z in Y", "there is no positive relation between X and z in Y".
- $\mathsf{Break}(X, Y)$ = True reads "there is a negative relation between the members of X in Y", "there is no negative relation between the members of X in Y" otherwise.

We want relations to reflect Domain Knowledge, in the sense that if some Domain expressions are inconsistent, then they are in a Break relation, while premises and a conclusion in a Domain proof are in a Build relation. This fits the rationale we gave above for Build and Break relations. Inconsistent Domain expressions cannot be true together, which in Problem-Solving Knowledge translates that they cannot all have positive statuses together. The analogy aplies to the Build relation as well: in

Problem-Solving Knowledge, Build ties the positive status of a sorted atom to positive statuses of other expressions, while in Domain Knowledge, proof ties the truth of the conclusion to the truth of the premises. This results in two rules below, which generate Build and Break relations from the proofs that the selector minproofs finds.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[Build]}, \mathsf{Rule[Break]}\}$$

where:

- Every minimal proof in Domain Knowledge is a Build relation in Problem-Solving Knowledge:

$$\begin{aligned}\mathsf{Rule[Build]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ \forall (Z, x) \in \mathsf{minproofs}(Y) \\ \neg(\mathsf{conclusion}(Z, x) = \bot) \Rightarrow \mathsf{Build}(\mathsf{premises}(Z, x), \mathsf{conclusion}(Z, x), Y)\end{aligned}$$

- Every minimal proof to inconsistency in Domain Knowledge is a Break relation in Problem-Solving Knowledge:

$$\begin{aligned}\mathsf{Rule[Break]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ \forall (Z, x) \in \mathsf{minproofs}(Y), \\ \mathsf{conclusion}(Z, x) = \bot \Rightarrow \mathsf{Break}(\mathsf{premises}(Z, x), Y)\end{aligned}$$

We said that a status of an atom depends on the statuses of other atoms and the relations it is in. We define these dependencies with the rules below. The overall idea is that the achievement of a goal depends on the execution of tasks and the maintenance of assumptions, and that execution and maintenance depend on the absence of break relations.

$$\begin{aligned}\mathsf{Rules} := \mathsf{Rules} \cup \{&\mathsf{Rule[Successful]}, \mathsf{Rule[Failed]}, \mathsf{Rule[Achieved]}, \mathsf{Rule[Maintained]} \\ &\mathsf{Rule[Executed]}\}\end{aligned}$$

where:

- An atom is successful if it has a positive status:

$$\begin{aligned}\mathsf{Rule[Successful]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}} \\ (\mathsf{Achieved}(x, Y) \vee \mathsf{Maintained}(x, Y) \vee \mathsf{Executed}(x, Y)) \\ \Leftrightarrow \mathsf{Succeeds}(x, Y) \\ \mathsf{Rule[Failed]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}} \\ (\neg\mathsf{Achieved}(x, Y) \vee \neg\mathsf{Maintained}(x, Y) \vee \neg\mathsf{Executed}(x, Y)) \\ \Leftrightarrow \neg\mathsf{Succeeds}(x, Y)\end{aligned}$$

- A goal x is achieved in the set Y iff (i) x is not in Break relations, and (ii) there is a good Build relation to x, in which all premises are successful. That is:

$$\begin{aligned}\mathsf{Rule[Achieved]} \equiv \forall Y, Z \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}} \\ \mathsf{G}(x, Y) \wedge \mathsf{BreakFree}(x, Y) \wedge \mathsf{GoodBuild}(x, Z, Y) \\ \wedge \forall w \in Z\ \mathsf{Succeeds}(w, Y)) \Leftrightarrow \mathsf{Achieved}(x, Y)\end{aligned}$$

- To be maintained, an assumption needs (i) to be in no Break relations, and either (ii-a) primitive, or, if it is not primitive, (ii-b) then there needs to be a good Build to it, in which all premises are successful. That is:

$$\begin{aligned}\mathsf{Rule[Maintained]} \equiv \forall Y, Z \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}}\\ \mathsf{A}(x, Y) \wedge \mathsf{BreakFree}(x, Y) \wedge (\mathsf{Primitive}(x, Y) \vee (\neg\mathsf{Primitive}(x, Y)\\ \wedge \mathsf{GoodBuild}(x, Z, Y) \wedge \forall w \in Z\ \mathsf{Succeeds}(w, Y)))\\ \Leftrightarrow \mathsf{Maintained}(x, Y)\end{aligned}$$

- To be executed, a task needs (i) to be in no Break relations, and either (ii-a) primitive, or, if it is not primitive, (ii-b) then there needs to be a good Build to it, in which all premises are successful. That is:

$$\begin{aligned}\mathsf{Rule[Executed]} \equiv \forall Y, Z \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}}\\ \mathsf{T}(x, Y) \wedge \mathsf{BreakFree}(x, Y) \wedge (\mathsf{Primitive}(x, Y) \vee (\neg\mathsf{Primitive}(x, Y)\\ \wedge \mathsf{GoodBuild}(x, Z, Y) \wedge \forall w \in Z\ \mathsf{Succeeds}(w, Y)))\\ \Leftrightarrow \mathsf{Executed}(x, Y)\end{aligned}$$

The TF specificaton above mentions the predicates for primitive expression, of being free of Break relations, and being in a good Build relation. We define these next.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[Primitive]}, \mathsf{Rule[BreakFree]}, \mathsf{Rule[GoodBuild]}\}$$

where:

- A Domain expression x is primitive in a set Y, denoted $\mathsf{Primitive}(x, Y)$, iff there is no Build relation to x from $Y \setminus \{x\}$:

$$\begin{aligned}\mathsf{Rule[Primitive]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ \forall x \in \mathcal{L}_{\text{CPL}}\\ (\nexists(Z, q) \in \mathsf{minproofs}(Y \setminus \{x\})\ \mathsf{Build}(Z, q, Y)\\ \wedge x = \mathsf{conclusion}(Z, q)\ ) \Leftrightarrow \mathsf{Primitive}(x, Y)\end{aligned}$$

- A Domain expression x is free of Break relations in a set Y, denoted $\mathsf{BreakFree}(x, Y)$, iff x participates in no Break relations:

$$\begin{aligned}\mathsf{Rule[BreakFree]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}},\ \forall x \in \mathcal{L}_{\text{CPL}}\\ (\nexists(Z, q) \in \mathsf{minproofs}(Y)\ \mathsf{Break}(Z, Y) \wedge x \in \mathsf{premises}(Z, q)\ )\\ \Leftrightarrow \mathsf{BreakFree}(x, Y)\end{aligned}$$

- A Domain expression x is said to have a good Build relation from Z in Y, denoted $\mathsf{GoodBuild}(x, Z, Y)$, iff x is the conclusion of a Build from Z, and every member of Z is successful in Y:

$$\begin{aligned}\mathsf{Rule[GoodBuild]} \equiv \forall Y \subseteq \mathcal{L}_{\text{CPL}}\ \forall x \in \mathcal{L}_{\text{CPL}}\\ (\exists(Z, x) \in \mathsf{minproofs}(Y) \wedge x = \mathsf{conclusion}(Z, x)\\ \wedge Z = \mathsf{premises}(Z, x)\ ) \wedge (\forall q \in Z\ \mathsf{Succeeds}(x, Y))\\ \Leftrightarrow \mathsf{GoodBuild}(x, Z, Y)\end{aligned}$$

We use no evaluations in **T1**; we will extend it with evaluations in Section 6. We finish the TF specification of **T1** with the problem and solution predicates.

$$
\begin{array}{l}
(\ \exists X \subseteq \mathcal{L}_{\text{CPL}}\ \exists z \in X\ \mathsf{G}(z, X) \wedge \neg \mathsf{Succeeds}(z, X)\ ) \\
\Leftrightarrow \mathsf{Problem}(X)
\end{array}
$$

$$
\begin{array}{l}
(\ \exists X \subseteq \mathcal{L}_{\text{CPL}}\ \neg \mathsf{Problem}(X) \wedge (\exists z \in X\ \mathsf{G}(z, X)) \\
\wedge \exists Y \subseteq X\ \forall z \in X\ (\mathsf{G}(z, X) \Rightarrow (\mathsf{G}(z, Y) \wedge \mathsf{Succeed}(z, Y)))\ ) \\
\Leftrightarrow \mathsf{Solution}(Y, X)
\end{array}
$$

The problem predicate says that there is a problem instance in a set X of Domain expressions iff there is a goal that fails. If there are goals in a set X and there is no problem instance in it, then there is a solution instance for that set X.

Table 2 gives a summary of the components of **T1** according to the TF specification we gave above.

Table 2: Summary of components in **T1**.

PART	COMPONENT	IN **T1**
D		*Domain Knowledge*, includes:
	$\mathcal{L}_{\mathbf{D}}$	$\mathcal{L}_{\text{CPL}}$
	$\mid\!\sim_{\mathbf{D}}$	$\vdash$
	X	$X \subseteq \mathcal{L}_{\text{CPL}}$
I		$\{\mathsf{atoms}(\cdot), \mathsf{minproofs}(\cdot), \mathsf{premises}(\cdot,\cdot), \mathsf{conclusion}(\cdot,\cdot)\}$
S		*Problem-Solving Knowledge*, includes:
	$\mathcal{L}_{\mathbf{S}}$	$\mathcal{L}_{\text{CFOL}}$
	$\mid\!\approx_{\mathbf{S}}$	$\models$
	$\langle X \rangle_{\mathbf{S}}$	$\langle X \rangle_{\mathbf{S}} \subseteq \mathcal{L}_{\text{CFOL}}$
	Sorts	$\{\mathsf{G}(\cdot,\cdot), \mathsf{A}(\cdot,\cdot), \mathsf{T}(\cdot,\cdot)\}$
	Statuses	$\{\mathsf{Achieved}(\cdot,\cdot), \mathsf{Maintained}(\cdot,\cdot), \mathsf{Executed}(\cdot,\cdot)\}$
	Relations	$\{\mathsf{Build}(\cdot,\cdot,\cdot), \mathsf{Break}(\cdot,\cdot)\}$
	Evaluations	*None.*
	Problem	$\mathsf{Problem}(\cdot)$
	Solution	$\mathsf{Solution}(\cdot,\cdot)$
	Rules	{Rule[GAT], Rule[Build], Rule[Break], Rule[Successful], Rule[Failed], Rule[Achieved], Rule[Maintained], Rule[Executed], Rule[Primitive], Rule[BreakFree], Rule[GoodBuild]}

5 RML Reconstruction

RML reconstruction is a particular case of RML specification, in which one specifies in TF an RML which was, when it was proposed, specified in some other way. It does not matter much how it was specified, as long as it was not with TF. We would want to do RML reconstruction when, for example, we want to compare a new RML with existing ones, so that we need to reconstruct the latter, or if we want to survey and compare existing RMLs.

The specification produced through RML reconstruction is necessarily debatable, as we have no criteria to say if the resulting specification captures correctly the RML as its authors intended it, and how much of it it manages to represent.

To illustrate RML reconstruction, we chose one existing RML, and used a single publication as the source of knowledge on that RML. The resulting specification is more interesting as an illuatration of TF, than as a comprehensive, valid, and definitive specification of the RML we selected.

We use TF to specify Yu's **i**∗ modeling language. We give three TF specifications for **i**∗ in the rest of this section, and refer to them as **iS1**, **iS2**, and **iS3**. All three are based on the early **i**∗ variant, which appeared at the *1st IEEE International Symposium on Requirements Engineering* [50]. All quotes in this section are from that publication, unless we give another reference. Other variants of **i**∗ appeared afterwards (e.g., [48]), but we do not consider them here. We will use **i**∗ to refer to the definition of that RML in the publication we selected, and **iS1**, **iS2**, and **iS3** to the TF specifications we make of it below.

The reason we make three specifications is to illustrate how different they can be, depending on decisions we make when using TF to specify **i**∗.

5.1 Background

i∗ is a modeling language that focuses on the representation of "organizational environments – an important class of environments within which many computer-based information systems operate".

An organization involves 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."

i∗ models are representations of information about goals of agents, plans that agents execute to achieve goals, tasks in plans, and resources used when executing tasks. It is agents who pursue goals, have plans, execute tasks, and deploy or furnish resources.

An important innovation in **i**∗ are the dependency relations between agents: an agent a may have a goal that she cannot or does not want to achieve by herself; another agent b may be able to achieve that goal, and if a delegates that goal to b, then a depends on b. Analogous dependency relations exist over the execution of tasks and the availability of resources.

An **i**∗ model will represent propositions about the dependencies between agents to have goals achieved, tasks executed, and resources made available.

Relations in **i**∗ are not only between propositions. Goal and task sort propositions, but agents and resources do not. Moreover, "in an organization, dependencies are usually not tied to a particular agent, but rather to a role. [...] We call a set of roles that are played by an agent a position." We thus need a language that can represent propositions, for goals and tasks, and constants for reference to agents, resources, roles, and positions. For example, "Mary has the role of project manager" is a proposition, but "Mary" and "project manager" are constants, referring respectively to an agent and a role, that **i**∗ represents in relation to that proposition.

5.2 iS1

Domain language for **iS1** consequently needs propositional variables and constants. And in Problem-Solving Knowledge, Agents, Roles, Positions, and Resources will be sorts over constants, while Goals and Tasks will be sorts over propositions.

An important observation is that **i**∗ has no notion of compositionality of propositions. This means that one cannot say, for example, that a goal proposition is a conjunction of some other, also sorted propositions, or that an agent is a part of another agent (which may be useful if we wanted to represent groups of agents, and allow agents to depend on groups). There is therefore no need to have a Domain formalism which allows the compositionality of propositions and constants.

There are different ways with TF to formalize the absence of compositionality in **iS1** Domain knowledge. For example, an option is to restrict Domain knowledge so that its language includes only a set of constants and a set of propositional variables. In that case, the consequence relation $\mathbf{\sim_D}$ would deduce, from any set of propositions, any one of these propositions, but nothing else. Another option would be to have a classical propositional language as $\mathcal{L}_{\mathbf{D}}$, and its consequence relation as $\mathbf{\vdash_D}$; but in that case, selectors would still need to return only propositional variables, and any formula could only be interesting if it is treated itself as a propositional variable in Problem-Solving Knowledge. If this second option is used, then the relations between a formula and a proposition that propositional logic gives (e.g., that p can be deduced from $p \wedge q$) are lost in Problem-Solving Knowledge: a selector would return p and $p \wedge q$ as two different and unrelated statements, both atomic as far as the RML is concerned.

We adopt the first option above, as it is simpler. This leads us to the following Domain Knowledge for **iS1**.

$\mathbf{D_{iS1}} = (\mathcal{L}_{\mathbf{iS1}}, \vdash_*, X)$, where:

- $\mathcal{L}_{\mathbf{iS1}} = \mathcal{P} \cup \mathcal{C}$, where $\mathcal{P}$ is a set of atoms, and $\mathcal{C}$ is a set of constants. Constants can be, for example, names of people, names of roles and positions, names of resources, etc.
- $\vdash_* \subseteq \wp(\mathcal{P}) \times \mathcal{P}$ is defined as follows: the atom $p \in \mathcal{P}$ is a consequence of $X \subseteq \mathcal{P}$, written $X \vdash_* p$ iff $p \in X$.
- X is some subset of $\mathcal{L}_{\mathbf{iS1}}$.

The Domain language distinguishes only between constants and atoms, and the Domain consequence relation relates uses only a single inference rule. We need only two selectors, one for propositions and another for constants.

$\mathbf{I_{iS1}} = \{\mathsf{constants}(\cdot), \mathsf{atoms}(\cdot)\}$, where:

- $\mathsf{constants}(Y)$ returns all constants in $Y \subseteq \mathcal{L}_{\mathbf{iS1}}$.
- $\mathsf{atoms}(Y)$ returns all atoms in $Y \subseteq \mathcal{L}_{\mathbf{iS1}}$.

We use CFOL as the formalism for Problem-Solving Knowledge, which gives the following first components of $\mathbf{S}$ for **iS1**.

$$\mathbf{S_{iS1}} = (\mathcal{L}_{\text{CFOL}}, \models, \langle X \rangle_{\mathbf{S}}, \mathsf{Sorts}, \mathsf{Statuses}, \mathsf{Relations}, \mathsf{Evaluations}, \mathsf{Problem}(\cdot), \mathsf{Solution}(\cdot), \mathsf{Rules})$$

where:

- $\mathcal{L}_{\text{CFOL}}$ is the language of CFOL.
- $\models$ is the satisfiability relation of CFOL.
- $\langle X \rangle_{\mathbf{S}}$ is a set of CFOL expressions. All constants in these expressions are members of Domain KB X, or of its closure.

To sort constants, Sorts gets the unary predicates $\mathsf{Agent}(\cdot)$, $\mathsf{Role}(\cdot)$, and $\mathsf{Resource}(\cdot)$. For example, $\mathsf{Agent}(a)$ is true iff $a \in \mathsf{constants}(X)$ refers to an Agent.

Sort predicates for constants are primitive and are always asserted by the modeler. Given some constant x, the modeler needs to add to Problem-Solving Knowledge that $\mathsf{Agent}(x)$, rather than deduce this from $\langle X \rangle_{\mathbf{S}}$. In other words, it will be the case that $\langle X \rangle_{\mathbf{S}} \models \mathsf{Agent}(a)$ iff $\mathsf{Agent}(a) \in \langle X \rangle_{\mathbf{S}}$.

A position is a set of roles an agent plays, so that position is a relation, not a sort. $\mathsf{Position}(R, a, p)$ is true iff p is the name of the position in which agent a plays the set of roles R. Names of positions are constants too, and we use the sort $\mathsf{PositionName}$ for these. The predicate $\mathsf{Plays}(a, r)$ is used to relate an agent a to a role r, that the agent plays.

The discussion above leads to the following update to $\mathbf{S_{iS1}}$.

$$
\begin{aligned}
\mathsf{Sorts} := \{&\mathsf{Agent} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\
&\mathsf{Role} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\
&\mathsf{Resource} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\
&\mathsf{PositionName} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\}\}\\
\mathsf{Relations} := \{&\mathsf{Plays} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\}\}\\
\mathsf{Rules} = \{&\mathsf{Rule[Position]}\}
\end{aligned}
$$

where:

- $\mathsf{Agent}(a) =$ True reads "a is an agent", "a is not an agent" otherwise.
- $\mathsf{Role}(a) =$ True reads "a is a role", "a is not a role" otherwise.
- $\mathsf{Resource}(a) =$ True reads "a is a resource", "a is not a resource" otherwise.
- $\mathsf{PositionName}(a) =$ True reads "a is a name of a position", "a is not a name of a position" otherwise.
- $\mathsf{Plays}(a, r) =$ True reads "agent a plays role r", "agent a does not play role r" otherwise.
- p is the name of the position in which an agent a plays all roles in R:

$$
\begin{aligned}
\mathsf{Rule[Position]} \equiv \exists a, p \in \mathsf{constants}(X)\ & R \subseteq \mathsf{constants}(X)\\
& (\mathsf{Agent}(a) \wedge \mathsf{PositionName}(p) \wedge (\forall r \in R\ \mathsf{Role}(r) \wedge \mathsf{Plays}(a, r)))\\
& \Leftrightarrow \mathsf{Position}(R, a, p)
\end{aligned}
$$

Goals and tasks are sorts over Domain atoms, and are specified with predicates $\mathsf{Goal}(\cdot)$ and $\mathsf{Task}(\cdot)$. For example, $\mathsf{Goal}(p)$ reads that proposition p is a goal.

$$
\begin{aligned}
\mathsf{Sorts} := \mathsf{Sorts} \cup \{&\mathsf{Goal} : \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\
&\mathsf{Task} : \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\}\}
\end{aligned}
$$

where:

- $\mathsf{Goal}(p) =$ True reads "p is a goal", "p is not a goal" otherwise.
- $\mathsf{Task}(p) =$ True reads "p is a task", "p is not a task" otherwise.

There are three dependency relations in **i**$*$:

- 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".

We define all three dependency relations in **iS1** using other primitive relations. We first introduce the rules that define the dependencies, and define these primitive relations then.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[GoalDependency]}, \mathsf{Rule[TaskDependency]}, \mathsf{Rule[ResourceDependency]}\}$$

where:

- That agent a depends on agent b to have goal x achieved means that a wants to have x achieved, cannot do it herself, and b is able to achieve x:

$$\begin{aligned}\mathsf{Rule[GoalDependency]} \equiv\ & \exists a, b, x\ \mathsf{Goal}(x) \wedge \mathsf{Agent}(a) \wedge \mathsf{WantsAchieved}(a, x) \\ & \wedge \neg \mathsf{CanAchieve}(a, x) \wedge \mathsf{Agent}(b) \wedge \mathsf{CanAchieve}(b, x) \\ & \Leftrightarrow \mathsf{GoalDependency}(a, b, x)\end{aligned}$$

- That agent a depends on agent b to have task x executed means that a needs to have x executed, cannot do it herself, and b is able to execute x:

$$\begin{aligned}\mathsf{Rule[TaskDependency]} \equiv\ & \exists a, b, x\ \mathsf{Task}(x) \wedge \mathsf{Agent}(a) \wedge \mathsf{WantsExecuted}(a, x) \\ & \wedge \neg \mathsf{CanExecute}(a, x) \wedge \mathsf{Agent}(b) \wedge \mathsf{CanExecute}(b, x) \\ & \Leftrightarrow \mathsf{TaskDependency}(a, b, x)\end{aligned}$$

- That agent a depends on agent b to make available resource x means that a needs to have access to x, cannot gain that access herself, and b can provide that access:

$$\begin{aligned}\mathsf{VC[ResourceDependency]} \equiv\ & \exists a, b, x\ \mathsf{Resource}(x) \wedge \mathsf{Agent}(a) \wedge \mathsf{WantsDelivered}(a, x) \\ & \wedge \neg \mathsf{CanDeliver}(a, x) \wedge \mathsf{Agent}(b) \wedge \mathsf{CanDeliver}(b, x) \\ & \Leftrightarrow \mathsf{ResourceDependency}(a, b, x)\end{aligned}$$

Yu's definition of **i**$*$ defines dependencies in a different way, as our relations in the specification fragment above differ from his. We will return to this issue later; for now, we only mention that main reason for this departure is that Yu defines dependencies in terms of agent's beliefs which we do not see as crucial to **iS1**. Once we leave agents' beliefs aside, we can have a simpler specification of **i**$*$, without losing much.

An atom can be either a goal or a task, never neither, nor both. This gives the following rule.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[GT]}\}$$

where Rule[GT] is that an atom is either a goal or a task:

$$\mathsf{Rule[GT]} \equiv \forall x \in \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}})(\mathsf{Goal}(x) \vee \mathsf{Task}(x)) \wedge \neg(\mathsf{Goal}(x) \wedge \mathsf{Task}(x))$$

We add a rule to define the dependum sort. It will help us shorten definitions of other rules below.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[Dependum]}\}$$

where Rule[Dependum] is that any goal, task, or resource x is a dependum if there is a dependency relation on it:

$$\begin{aligned}\mathsf{Rule[Dependum]} \equiv \forall x\ (&\exists a, b\ \mathsf{GoalDependency}(a, b, x)\\ &\vee \mathsf{TaskDependency}(a, b, x) \vee \mathsf{ResourceDependency}(a, b, x))\\ &\Leftrightarrow \mathsf{Dependum}(x)\end{aligned}$$

Pursuing, then, with the specification of our understanding of $\mathbf{i}*$, we consider as primitive all predicates used in rules that define dependency relations. This leads to the following update to $\mathbf{S_{iS1}}$.

$$\begin{aligned}\mathsf{Relations} := \mathsf{Relations} \cup \{&\mathsf{WantsAchieved} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{CanAchieve} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{WantsExecuted} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{CanExecute} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{WantsDelivered} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\},\\ &\mathsf{CanDeliver} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}}) \longrightarrow \{\text{True}, \text{False}\}\}\end{aligned}$$

where:

- $\mathsf{WantsAchieved}(a, x)$ = True reads "agent a wants goal x achieved", "agent a does not want goal x achieved" otherwise.
- $\mathsf{CanAchieve}(a, x)$ = True reads "agent a can achieve goal x", "agent a cannot achieve goal x" otherwise.
- $\mathsf{WantsExecuted}(a, x)$ = True reads "agent a wants task x executed", "agent a does not want task x executed" otherwise.
- $\mathsf{CanExecute}(a, x)$ = True reads "agent a can execute task x", "agent a cannot execute task x" otherwise.
- $\mathsf{WantsDelivered}(a, x)$ = True reads "agent a wants resource x delivered", "agent a does not want resource x delivered" otherwise.
- $\mathsf{CanDeliver}(a, x)$ = True reads "agent a can deliver resource x", "agent a cannot deliver resource x" otherwise.

There are no evaluations in $\mathbf{i}*$, leaving Evaluations empty.

$$\mathsf{Evaluations} := \emptyset$$

If we consider that a goal will be achieved if there is an agent who wants it achieved, and another who can achieve it, we can use the following definition of the status achieved. It is straightforward to define by analogy the statuses executed for tasks, and delivered for resources.

$$\begin{aligned}\mathsf{Statuses} := \{&\mathsf{Achieved} : \{x \mid \mathsf{Goal}(x)\} \longrightarrow \{\mathrm{True}, \mathrm{False}\},\\ &\mathsf{Executed} : \{x \mid \mathsf{Task}(x)\} \longrightarrow \{\mathrm{True}, \mathrm{False}\},\\ &\mathsf{Delivered} : \{x \mid \mathsf{Resource}(x)\} \longrightarrow \{\mathrm{True}, \mathrm{False}\}\}\\ \mathsf{Rules} := \mathsf{Rules} \cup \{&\mathsf{Rule[Achieved]}, \mathsf{Rule[Executed]}, \mathsf{Rule[Delivered]}, \mathsf{Rule[Successful]}, \mathsf{Rule[Failed]}\}\end{aligned}$$

where:

- $\mathsf{Achieved}(x) = \mathrm{True}$ reads "goal x is achieved", "goal x is not achieved" otherwise.
- $\mathsf{Executed}(x) = \mathrm{True}$ reads "task x is executed", "task x is not executed" otherwise.
- $\mathsf{Delivered}(x) = \mathrm{True}$ reads "resource x is delivered", "resource x is not delivered" otherwise.
- A goal is achieved if there is an agent who wants it achieved and another who can achieve it:

$$\mathsf{Rule[Achieved]} \equiv \forall x \in \mathcal{L}_{\mathbf{iS1}}\ \mathsf{Goal}(x) \wedge \mathsf{Dependum}(x) \Leftrightarrow \mathsf{Achieved}(x)$$

- A task is executed if there is an agent who wants it executed and another who can execute it:

$$\mathsf{Rule[Executed]} \equiv \forall x \in \mathcal{L}_{\mathbf{iS1}}\ \mathsf{Task}(x) \wedge \mathsf{Dependum}(x) \Leftrightarrow \mathsf{Executed}(x)$$

- A resource is delivered if there is an agent who wants it delivered and another who can deliver it:

$$\mathsf{Rule[Delivered]} \equiv \forall x \in \mathcal{L}_{\mathbf{iS1}}\ \mathsf{Resource}(x) \wedge \mathsf{Dependum}(x) \Leftrightarrow \mathsf{Delivered}(x)$$

- A goal, task, or resource is successful if it is, respectively, achieved, executed, or delivered:

$$\begin{aligned}\mathsf{Rule[Successful]} \equiv \forall x \in \mathcal{L}_{\mathbf{iS1}}&\\ &(\mathsf{Goal}(x) \wedge \mathsf{Achieved}(x)) \vee (\mathsf{Task}(x) \wedge \mathsf{Executed}(x))\\ &\vee (\mathsf{Resource}(x) \wedge \mathsf{Delivered}(x)) \Leftrightarrow \mathsf{Succeeds}(x)\end{aligned}$$

- A goal, task, or resource fails if it is, respectively, not achieved, not executed, or not delivered:

$$\begin{aligned}\mathsf{Rule[Failed]} \equiv \forall x \in \mathcal{L}_{\mathbf{iS1}}&\\ &(\mathsf{Goal}(x) \wedge \neg\mathsf{Achieved}(x)) \vee (\mathsf{Task}(x) \wedge \neg\mathsf{Executed}(x))\\ &\vee (\mathsf{Resource}(x) \wedge \neg\mathsf{Delivered}(x)) \Leftrightarrow \neg\mathsf{Succeeds}(x)\end{aligned}$$

Problem and solution concepts are the following ones.

$$\exists X \subseteq \mathcal{L}_{\mathbf{iS1}}\ \exists z \in X\ (\mathsf{Goal}(z) \vee \mathsf{Task}(z) \vee \mathsf{Resource}(z)) \wedge \neg\mathsf{Succeeds}(z) \Leftrightarrow \mathsf{Problem}(X)$$
$$\exists X \subseteq \mathcal{L}_{\mathbf{iS1}}\ \neg\mathsf{Problem}(X) \Leftrightarrow \mathsf{Solution}(X)$$

The problem above says that if we have an **iS1** model in which there is some goal, task, or resource, and it fails, then there is a problem with that model. As soon as all goals, tasks, and resources in a model are successful, that model solves the problem.[2]

Table 3 gives a summary of the components of **iS1** according to the TF specification we gave above.

Table 3: Summary of components in **iS1**.

Part	Component	In **iS1**
D		*Domain Knowledge*, includes:
	$\mathcal{L}_{\mathsf{D}}$	$\mathcal{L}_{\mathbf{iS1}}$
	$\mid\!\sim_{\mathsf{D}}$	$\mid\!\sim_{*}$
	X	$X \subseteq \mathcal{L}_{\mathbf{iS1}}$
I		$\{\mathsf{constants}(\cdot), \mathsf{atoms}(\cdot)\}$
S		*Problem-Solving Knowledge*, includes:
	$\mathcal{L}_{\mathsf{S}}$	$\mathcal{L}_{\text{CFOL}}$
	$\mid\!\approx_{\mathsf{S}}$	$\models$
	$\langle X \rangle_{\mathsf{S}}$	$\langle X \rangle_{\mathsf{S}} \subseteq \mathcal{L}_{\text{CFOL}}$
	Sorts	$\{\mathsf{Agent}(\cdot), \mathsf{Role}(\cdot), \mathsf{Resource}(\cdot), \mathsf{PositionName}(\cdot), \mathsf{Goal}(\cdot), \mathsf{Task}(\cdot)\}$
	Statuses	$\{\mathsf{Achieved}(\cdot), \mathsf{Executed}(\cdot), \mathsf{Delivered}(\cdot)\}$
	Relations	$\{\mathsf{Plays}(\cdot,\cdot), \mathsf{WantsAchieved}(\cdot,\cdot), \mathsf{CanAchieve}(\cdot,\cdot), \mathsf{WantsExecuted}(\cdot,\cdot),$ $\mathsf{CanExecute}(\cdot,\cdot), \mathsf{WantsDelivered}(\cdot,\cdot), \mathsf{CanDeliver}(\cdot,\cdot)\}$
	Evaluations	*None.*
	Problem	$\mathsf{Problem}(\cdot)$
	Solution	$\mathsf{Solution}(\cdot,\cdot)$
	Rules	{Rule[Position], Rule[GoalDependency], Rule[TaskDependency], Rule[ResourceDependency], Rule[GT], Rule[Dependum], Rule[Achieved], Rule[Executed], Rule[Delivered], Rule[Successful], Rule[Failed]}

[2]There is a way to define the problem and solution in **iS1** without using statuses. Notice that success equates with being in a dependency relation, in Rule[Achieved], Rule[Executed], Rule[Delivered]. The simpler problem and solution are therefore as follows.

$$\exists X \subseteq \mathcal{L}_{\mathbf{iS1}}\ \exists z \in X\ (\mathsf{Goal}(z) \vee \mathsf{Task}(z) \vee \mathsf{Resource}(z)) \wedge \neg\mathsf{Dependum}(z) \Leftrightarrow \mathsf{Problem}(X)$$
$$\exists X \subseteq \mathcal{L}_{\mathbf{iS1}}\ \neg\mathsf{Problem}(X) \Leftrightarrow \mathsf{Solution}(X)$$

5.3 iS2

Yu emphasized that dependencies are intentional relationships, meaning that they are defined in terms of agents' beliefs. This is not reflected at all in **iS1**. Yu's own axiomatization uses a first-order logic with the belief modality, which we will not do here. We want to keep CFOL as the Problem-Solving formalism, so that we will not use a belief modality, but instead, new relations that refer to intentions.

To have a specification of **i**$*$ that defines dependencies in terms of belief, desire and intention, we define a new RML and call it **iS2**.

iS2 is **iS1** with one change: in **iS2**, $\mathsf{WantsAchieved}(\cdot,\cdot)$, $\mathsf{CanAchieve}(\cdot,\cdot)$, $\mathsf{WantsExecuted}(\cdot,\cdot)$, $\mathsf{CanExecute}(\cdot,\cdot)$, $\mathsf{WantsDelivered}(\cdot,\cdot)$, and $\mathsf{CanDeliver}(\cdot,\cdot)$ are *not* primitive. Instead, we add rules which define them via belief, desire, and intention relations, also new compared to **iS1**.

To define **iS2**, we first let **iS2** := **iS1**, so that it includes all **iS1** did, and we progressively make changes below.

We read from $\mathsf{WantsAchieved}(a,x)$ that agent a wants to have goal x achieved. For Yu [50] it would mean that "a believes that it has some plan p for achieving some goal ϕ_0 which contains an activity α which has x as an external subgoal". Our understanding of this is that a has some other goal y, a plan p to achieve y, that x is one of the subgoals in the plan, and that a delegates the achievement of x. We use the following definition in **iS2**.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[WantsAchieved]}\}$$

where:

- $\mathsf{WantsAchieved}(a,x)$ means that (i) a intends some other goal y, (ii) plan p is for achieving y, (iii) a intends p, (iv) goal x is part of the plan p, (v) a delegates the achievement of x:

$$\begin{aligned}\mathsf{Rule[WantsAchieved]} \equiv \forall a,x\ \mathsf{WantsAchieved}(a,x) \Leftrightarrow\ & \mathsf{Goal}(x) \wedge \mathsf{Agent}(a)\\ & \wedge (\exists y,p\ \mathsf{Goal}(y) \wedge \mathsf{Plan}(p,y)) \wedge \mathsf{Intend}(a,y) \wedge \mathsf{Intend}(a,p)\\ & \wedge \mathsf{PartOf}(x,p) \wedge \mathsf{Delegate}(a,x)\end{aligned}$$

We used the plan sort, and the intend, part of, and delegate relations above, all of which are new in **iS2** compared to **iS1**. We add them as follows.

$$\begin{aligned}\mathsf{Sorts} &:= \mathsf{Sorts} \cup \{\mathsf{Plan} : \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \longrightarrow \{\mathrm{True},\mathrm{False}\}\}\\ \mathsf{Relations} &:= \mathsf{Relations} \cup \{\mathsf{Intend} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS2}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \longrightarrow \{\mathrm{True},\mathrm{False}\},\\ &\qquad \mathsf{PartOf} : \{x \in \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \mid \forall y\ \neg\mathsf{Plan}(x,y)\}\\ &\qquad\quad \times \{z \in \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \mid \exists w\ \mathsf{Plan}(z,w)\} \longrightarrow \{\mathrm{True},\mathrm{False}\},\\ &\qquad \mathsf{Delegate} : \{x \in \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \mid \forall y\ \neg\mathsf{Plan}(x,y)\}\\ &\qquad\quad \times \{a \mid \mathsf{Agent}(a)\} \longrightarrow \{\mathrm{True},\mathrm{False}\}\}\end{aligned}$$

where:

- $\mathsf{Plan}(p,y) = \mathrm{True}$ reads "p is a plan for satisfying y", "p is not a plan for satisfying y" otherwise.
- $\mathsf{Intend}(a,y) = \mathrm{True}$ reads "a commits to y", "a does not commit to y" otherwise.
- $\mathsf{PartOf}(x,p)$ reads "x is part of plan p", "x is not part of plan p" otherwise.

- $\mathsf{Delegate}(a, x)$ reads "a delegates x to another agent", "a does not delegate x to another agent" otherwise.

By analogy to Rule[WantsAchieved], we define $\mathsf{WantsExecuted}(\cdot,\cdot)$ and $\mathsf{WantsDelivered}(\cdot,\cdot)$.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[WantsExecuted]}, \mathsf{Rule[WantsDelivered]}\}$$

where:

- $\mathsf{WantsExecuted}(a, x)$ means that (i) a intends some goal y, (ii) plan p is for achieving y, (iii) a intends p, (iv) task x is part of the plan p, (v) a delegates the execution of x:

$$\begin{aligned}\mathsf{Rule[WantsExecuted]} \equiv \forall a, x\ \mathsf{WantsExecuted}(a, x) \Leftrightarrow\ & \mathsf{Task}(x) \wedge \mathsf{Agent}(a) \\ & \wedge (\exists y, p\ \mathsf{Goal}(y) \wedge \mathsf{Plan}(p, y)) \wedge \mathsf{Intend}(a, y) \wedge \mathsf{Intend}(a, p) \\ & \wedge \mathsf{PartOf}(x, p) \wedge \mathsf{Delegate}(a, x)\end{aligned}$$

- $\mathsf{WantsDelivered}(a, x)$ means that (i) a intends some goal y, (ii) plan p is for achieving y, (iii) a intends p, (iv) resource x is needed for the plan p, (v) a delegates the delivery of x:

$$\begin{aligned}\mathsf{Rule[WantsDelivered]} \equiv \forall a, x\ \mathsf{WantsDelivered}(a, x) \Leftrightarrow\ & \mathsf{Resource}(x) \wedge \mathsf{Agent}(a) \\ & \wedge (\exists y, p\ \mathsf{Goal}(y) \wedge \mathsf{Plan}(p, y)) \wedge \mathsf{Intend}(a, y) \wedge \mathsf{Intend}(a, p) \\ & \wedge \mathsf{PartOf}(x, p) \wedge \mathsf{Delegate}(a, x)\end{aligned}$$

In a dependency, the counterpart to $\mathsf{WantsAchieved}(a, x)$ is $\mathsf{CanAchieve}(b, x)$. In $\mathbf{i}*$, this counterpart conveys the ability of agent b to achieve the goal in the dependency, but not that agent's intention to do so; same applies, by analogy, to the execution of tasks and the delivery of resources: "an agent is able to achieve [a goal] if it believes it has some plan which will result in [that goal] being true. An agent can perform [here, execute] plan / activity if [it] is in [that agent's] repertoire. An agent can furnish [a] resource if it has a plan which results in [that resource] being available." For us, an agent can achieve a goal if it has a plan for it.

We define $\mathsf{CanAchieve}(b, x)$, $\mathsf{CanExecute}(b, x)$, and $\mathsf{CanDeliver}(b, x)$ as follows.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[CanAchieve]}, \mathsf{Rule[CanExecute]}, \mathsf{Rule[CanDeliver]}\}$$
$$\mathsf{Relations} := \mathsf{Relations} \cup \{\mathsf{Intend} : \mathsf{constants}(\mathcal{L}_{\mathbf{iS2}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{iS2}}) \longrightarrow \{\mathrm{True}, \mathrm{False}\}\}$$

where:

- $\mathsf{CanAchieve}(b, x)$ means that (i) agent b has plan p, and (ii) p is a plan for achieving the goal x.

$$\begin{aligned}\mathsf{Rule[CanAchieve]} \equiv \forall b, x\ \mathsf{CanAchieve}(b, x)\ & \\ \Leftrightarrow\ & \mathsf{Goal}(x) \wedge \mathsf{Agent}(b) \wedge \mathsf{Plan}(p, x) \wedge \mathsf{CanIntend}(b, p)\end{aligned}$$

- $\mathsf{CanExecute}(b, x)$ means that (i) agent b has plan p, and (ii) p is a plan for executing the task x.

$$\begin{aligned}\mathsf{Rule[CanExecute]} \equiv \forall b, x\ \mathsf{CanExecute}(b, x)\ & \\ \Leftrightarrow\ & \mathsf{Task}(x) \wedge \mathsf{Agent}(b) \wedge \mathsf{Plan}(p, x) \wedge \mathsf{CanIntend}(b, p)\end{aligned}$$

- $\mathsf{CanDeliver}(b, x)$ means that (i) agent b has plan p, and (ii) p is a plan for delivering the resource x.

$$\mathsf{Rule[CanDeliver]} \equiv \forall b, x\ \mathsf{CanDeliver}(b, x) \Leftrightarrow \mathsf{Resource}(x) \wedge \mathsf{Agent}(b) \wedge \mathsf{Plan}(p, x) \wedge \mathsf{CanIntend}(b, p)$$

- $\mathsf{CanIntend}(b, p)$ reads "agent b can choose to commit to p", "agent b cannot choose to commit to p" otherwise.

iS2 differs from **iS1** in having more predicates and sorts, and rules that together define relations which we took for primitive in **iS1**. These relations were used in **iS1** to define dependencies. By redefining them in **iS2** with $\mathsf{Intend}(\cdot, \cdot)$ and $\mathsf{CanIntend}(\cdot, \cdot)$, we introduced a notion of intention. This remains a fairly simple way to introduce intentional notions; a more interesting, but also more complicated way, cosists of providing also rules needed to deduce these predicates, rather than have to assert them – as they remain primitive, and as no rules other than the new ones in **iS2** mentions them, we can only assert them in the Problem-Solving KB.

5.4 iS3

While **iS2** introduced a notion of intention, it failed to capture the distinction between dependencies in which there is no commitment from the dependee, and those where such commitment is present. This is important if we want to have a solution concept, in which the dependee needs to commit to the delegated goal, task, or resource.

We can accomplish this by defining **iS3** by defining new kinds of dependency relations, in which the depender intends to achieve, execute, or deliver, respectively, the goal, task, or resource.

Doing this is straightforward, as **iS2** has the predicate $\mathsf{Intend}(\cdot, \cdot)$. Let **iS3** := **iS2**, and add the following rules to **iS3**.

$$\mathsf{Rules}_{\mathbf{iS3}} := \mathsf{Rules}_{\mathbf{iS3}} \cup \{\mathsf{Rule[CommittedAchieve]}, \mathsf{Rule[CommittedExecute]}, \mathsf{Rule[CommittedDeliver]}\}$$

where:

- $\mathsf{CommittedAchieve}(b, x)$ means that (i) agent b has plan p, (ii) p is a plan for achieving the goal x, and (iii) a intends p.

$$\mathsf{Rule[CommittedAchieve]} \equiv \forall b, x\ \mathsf{CommittedAchieve}(b, x) \Leftrightarrow \mathsf{Goal}(x) \wedge \mathsf{Agent}(b) \wedge \mathsf{Plan}(p, x) \wedge \mathsf{Intend}(b, p)$$

- $\mathsf{CommittedExecute}(b, x)$ means that (i) agent b has plan p, (ii) p is a plan for executing the task x, and (iii) a intends p.

$$\mathsf{Rule[CommittedExecute]} \equiv \forall b, x\ \mathsf{CommittedExecute}(b, x) \Leftrightarrow \mathsf{Task}(x) \wedge \mathsf{Agent}(b) \wedge \mathsf{Plan}(p, x) \wedge \mathsf{Intend}(b, p)$$

- $\mathsf{CommittedDeliver}(b, x)$ means that (i) agent b has plan p, (ii) p is a plan for delivering the resource x, and (iii) a intends p.

$$\mathsf{Rule[CommittedDeliver]} \equiv \forall b, x\ \mathsf{CommittedDeliver}(b, x) \Leftrightarrow \mathsf{Resource}(x) \wedge \mathsf{Agent}(b) \wedge \mathsf{Plan}(p, x) \wedge \mathsf{Intend}(b, p)$$

The rules above define a variant of achieve, execute, and deliver, where the agent also intends to perform the plan which will result in the achievement, execution, or delivery. These rules give us predicates which we use to define the following variants of each dependency relation that we had in **iS1** and **iS2**.

$$\mathsf{Rules}_{\mathbf{iS3}} := \mathsf{Rules}_{\mathbf{iS3}} \cup \{\mathsf{Rule[CGoalDependency]}, \mathsf{Rule[CTaskDependency]}, \mathsf{Rule[CResourceDependency]}\}$$

where:

- That agent a depends on the commitment of agent b to have goal x achieved means that a wants to have x achieved, cannot do it herself, and b commits to achieve x:

$$\begin{aligned}\mathsf{Rule[CGoalDependency]} \equiv \exists a, b, x\ & \mathsf{Goal}(x) \wedge \mathsf{Agent}(a) \wedge \mathsf{WantsAchieved}(a, x) \\ & \wedge \neg \mathsf{CanAchieve}(a, x) \wedge \mathsf{Agent}(b) \wedge \mathsf{CommittedAchieve}(b, x) \\ & \Leftrightarrow \mathsf{CGoalDependency}(a, b, x)\end{aligned}$$

- That agent a depends on the commitment of agent b to have task x executed means that a needs to have x executed, cannot do it herself, and b commits to execute x:

$$\begin{aligned}\mathsf{Rule[CTaskDependency]} \equiv \exists a, b, x\ & \mathsf{Task}(x) \wedge \mathsf{Agent}(a) \wedge \mathsf{WantsExecuted}(a, x) \\ & \neg \mathsf{CanExecute}(a, x) \wedge \mathsf{Agent}(b) \wedge \mathsf{CommittedExecute}(b, x) \\ & \Leftrightarrow \mathsf{CTaskDependency}(a, b, x)\end{aligned}$$

- That agent a depends the commitment of agent b to make available resource x means that a needs to have access to x, cannot gain that access herself, and b commits to provide that access:

$$\begin{aligned}\mathsf{VC[CResourceDependency]} \equiv \exists a, b, x\ & \mathsf{Resource}(x) \wedge \mathsf{Agent}(a) \wedge \mathsf{WantsDelivered}(a, x) \\ & \neg \mathsf{CanDeliver}(a, x) \wedge \mathsf{Agent}(b) \wedge \mathsf{CommittedDeliver}(b, x) \\ & \Leftrightarrow \mathsf{CResourceDependency}(a, b, x)\end{aligned}$$

We can use the new dependency relations to redefine in **iS3** the predicate $\mathsf{Dependum}(\cdot)$ via these committed dependencies. The consequence of this is that the solution will require committted dependencies, not the original ones from **iS1** and **iS2**. This redefined $\mathsf{Dependum}(\cdot)$ is the following one. Rule[Dependum] below replaces that original rule from **iS1**.

Rule[Dependum] is that any goal, task, or resource x is a dependum if there is a committed dependency relation on it:

$$\begin{aligned}\mathsf{Rule[Dependum]} \equiv \forall x \in \mathsf{atoms}(\mathcal{L}_{\mathbf{iS1}})\ & (\exists a, b\ \mathsf{CGoalDependency}(a, b, x) \\ & \vee \mathsf{CTaskDependency}(a, b, x) \vee \mathsf{CResourceDependency}(a, b, x)) \\ & \Leftrightarrow \mathsf{Dependum}(x)\end{aligned}$$

Another way to achieve the same effect would be to leave the definition of Rule[Dependum] from **iS1** and **iS2**, and define a new predicate $\mathsf{CommittedDependum}(\cdot)$ and the Rule[CommittedDependum].

This would require further changes to rules Rule[Achieved], Rule[Executed], and Rule[Delivered], and the problem and solution concepts, where every occurrence of Dependum(·) would be replaced by CommittedDependum(·).

6 RML Change

RML change consists of making a new TF specification by changing one or more components of an existing specification.

We categorize changes according to the components that they modify in a TF specification. Given that a TF specification includes formal languages, inference rules, and predicates, it is not feasible to define all possible types of changes we can make. Instead, we discuss and illustrate several of these changes below, at least one for each component in TF specifications.

6.1 Domain Formalism Replacement

Domain formalism replacement consists of replacing the Domain language $\mathcal{L}_{\mathbf{D}}$ and the Domain consequence relation by another one.

Suppose that temporal properties of a system-to-be are an important consideration in stakeholders' requirements. This is the case in ambulance dispatching systems, as illustrated in the London Ambulance Service [1] case. They need to satisfy such requirements as that the ambulance needs to arrive at an incident location within some given number of minutes. In that case, the readability of classical propositional logic expressions in **D** of **T1** become a drawback, rather than an advantage.

If **S** in **T1** still seems useful, we can take **T1** and replace its $\mathcal{L}_{\text{CPL}}$ with $\mathcal{L}_{LTL}$ and $\vdash$ with $\models_{LTL}$, where $\mathcal{L}_{LTL}$ and $\models_{LTL}$ refer to, respectively, the language and satisfiability relation of first-order linear temporal logic. Let **T1t** refer to the new RMLSpec made by this change from **T1**; we will use it below to illustrate other RML Change types.

6.2 Domain Language Replacement

Domain language replacement consists of replacing the Domain language $\mathcal{L}_{\mathbf{D}}$ with another one, while keeping the Domain consequence relation unchanged.

To be useful, the replacement language has to fit the existing consequence relation. This will be the case if, for example, the replacement language uses the same inference rules, or only a subset of them. For illustration, take **T1**, and replace the Domain language with the one generated by these BNF rules:

$$\begin{aligned} \mathit{atom} &::= p \mid q \mid r \mid s \mid \ldots \\ \mathit{formula} &::= \mathit{atom}_1 \wedge \mathit{atom}_2 \wedge \ldots \wedge \mathit{atom}_{n-1} \rightarrow \mathit{atom}_n \\ &\quad \mid \mathit{atom}_1 \wedge \mathit{atom}_2 \wedge \ldots \wedge \mathit{atom}_n \rightarrow \bot \end{aligned}$$

The language above is a subset of the Domain language in **T1**, as every expression in it is also an expression in $\mathcal{L}_{\text{CPL}}$. One motive to use this language, instead $\mathcal{L}_{\text{CPL}}$ is to intentionally restrict the range of expressions that can be in the Domain KB, perhaps because this language may be easier for RML users to learn.

Another case when Domain language replacement may be relevant is in projects where we have a domain ontology [33], as the definition of concepts and relations that are key to understand the domain. We may want to restrict all expressions to only those which use those concepts and relations from the domain ontology. We would want to do this in order to, for example, avoid terminology clashes [45], which are the use of different names for the same concepts or relations in the domain. We could do this by taking a first-order language for the Domain language, and restrict predicates to only those referring to concepts or concept properties, and those referring to relations.

6.3 Domain Inference Rules Replacement

Domain inference rules replacement, also consequence relation replacement, consists of replacing the inference rules in $\mathbf{D}$ by another set of rules, and thereby replace the consequence relation, while keeping the Domain language unchanged.

Inference rules replacement can be used to allow or disable specific proof patterns. For example, inference rules may allow *ex falso quodlibet* proof pattern, according to which anything can be deduced from an inconsistent set of expressions. This is the case in classical propositional logic, and so in **T1**. It makes the Domain KB X explode, in the sense that the closure of X will be equal to $\mathcal{L}_{\text{CPL}}$ in **T1**, whenever $X \vdash \bot$.

If we want to avoid ex falso quodlibet, but keep the language of classical propositional logic for $\mathcal{L}_{\mathbf{D}}$, we can replace the inference rules that define $\vdash$ with inference rules from Besnard and Hunter's quasi-classical logic [3], for example. Let **T1qc** refer to the TF specification made by taking **T1** and making only this change of inference rules to it. Because of this, **T1qc** has a paraconsistent Domain consequence relation, and it will not have an exploding Domain KB that **T1** has.

6.4 Interface Replacement

Interface change consists of replacing original selectors with new ones. Such changes will influence which formulas from Domain KB, or of its closure, can appear as terms in the Problem-Solving KB.

We imposed no constraints on selectors in a TF specification, so that we can define them to work in different ways. They can return Domain formulas or proofs, they can return formulas having some syntactic shape, such as only atoms, or formulas that are conclusions from proofs, where the proof satisfies some properties, such as having the minimal set of premises to derive its conclusion. A selector could also be defined to return only proofs in which a single inference rule is applied. If Domain language is propositional, it may be useful to have a selector that returns all proper nouns in propositions, so that we can use these names for agents, something we could have used in **iS1**.

In **T1**, we had selectors that fed atoms into sorts, as well as the selector $\mathsf{minproofs}(\cdot)$ which instead looked at premises and conclusions in proofs from Domain KB, and returned only those proofs which were minimal with regards to deriving the conclusion. If we anticipated that the **T1** Domain KBs can be inconsistent, perhaps we would want to replace $\mathsf{minproofs}(\cdot)$ with a selector $\mathsf{minmaxconsproofs}(\cdot)$, which would return only those minimal proofs where premises are members of maximally consistent subsets of the Domain KB.

6.5 Specialization

Given two TF specifications A and B, we say that A is a specialization of B if B includes all sorts, statuses, relations, and evaluations of A and in addition includes Sorts, $\mathsf{Statuses}$, $\mathsf{Relations}$, or $\mathsf{Evaluations}$ that specialize those in A.

For illustration, suppose that the goal sort in **T1** is specialized onto organizational goal and personal goal. The aim could be to distinguish the official goals that an organization has for the system-to-be, from those that individuals may have, who will be using that system-to-be. This may be useful if we wanted to analyzed the fit or departure between organizational and personal goals, and use the conclusions of such an analysis to choose the security features of the system-to-be.

There are different ways in which we could specify our adding of organizational and personal goal sorts. For example, we would add predicates $\mathsf{OG}(\cdot,\cdot)$ for the organizational goal sort, and $\mathsf{PG}(\cdot,\cdot)$ for the personal goal sort, so that the new Sorts set would be as follows.

$$\mathsf{Sorts} = \{\mathsf{G}(\cdot,\cdot), \mathsf{A}(\cdot,\cdot), \mathsf{T}(\cdot,\cdot), \mathsf{PG}(\cdot,\cdot), \mathsf{OG}(\cdot,\cdot)\}$$

We would also need rules that capture the idea that organizational and personal goals are types of goal.

$$\mathsf{Rules} := \mathsf{Rules} \cup \{\mathsf{Rule[OGG]}, \mathsf{Rule[PGG]}, \mathsf{Rule[OGPG]}\}$$

where:

- Every organizational goal is also a goal:

$$\mathsf{Rule[OGG]} \equiv \ \forall X \subseteq \mathcal{L}_{\text{CPL}}\ z \in X\ \mathsf{OG}(z, X) \Rightarrow \mathsf{G}(z, X)$$

- Every personal goal is also a goal:

$$\mathsf{Rule[PGG]} \equiv \ \forall X \subseteq \mathcal{L}_{\text{CPL}}\ z \in X\ \mathsf{PG}(z, X) \Rightarrow \mathsf{G}(z, X)$$

- Every goal is either an organizational goal or a personal goal:

$$\begin{aligned}\mathsf{Rule[OGPG]} \equiv \forall X \subseteq \mathcal{L}_{\text{CPL}}\ \forall z \in X \\ \mathsf{G}(z, X) \Rightarrow ((\mathsf{OG}(z, X) \vee \mathsf{PG}(z, X)) \wedge \neg(\mathsf{OG}(z, X) \wedge \mathsf{PG}(z, X)))\end{aligned}$$

Let **T1op** be the name of the TF specification created by adding the sorts and rules above to **T1**. **T1op** lets us represent organizational and personal goals separately.

6.6 Evaluation Introduction

The introduction of evaluations amounts to adding new evaluation relations to Evaluations, some new Rules as needed to define how the evaluations work, and if needed, changing the rest of the TF specification (for example, to take evaluations into account in the problem or solution concepts).

We take **T1** for illustration. Its Evaluations set is empty. In RE, one of the important ideas is that of refinement: to refine a requirement, we add new requirements which are more detailed than the former. We say that the latter refine the former.

To add refinement to **T1**, we make a new RMLSpec, denote it **T1r** and let it be identical to **T1** for now: **T1r** := **T1**.

To capture the idea of refinement of goals, assumptions, and tasks in **T1r**, we need a partial order over Domain expressions. That partial order should compare Domain expressions in terms of their level of detail. It is not a total order, because many Domain expressions cannot be compared in terms of level of detail.

We use the predicate $\mathsf{geqDetail}(x, z, Y)$, which reads that x is at least as detailed as z in Y. In addition, we add Rules to ensure it is a partial order: that it is reflexive, antisymmetric, and transitive.

$$\mathsf{Evaluations}_{\mathbf{T1r}} := \{\mathsf{geqDetail} : \mathsf{atoms}(\mathcal{L}_{\text{CPL}}) \times \mathsf{atoms}(\mathcal{L}_{\text{CPL}}) \times \wp(\mathcal{L}_{\text{CPL}}) \longrightarrow \{\text{True}, \text{False}\}\}$$

where $\mathsf{geqDetail}(x, y, Z) = \text{True}$ reads "x is at least as detailed as y in Z", "x is not at least as detailed as y in Z" otherwise.

$$\begin{aligned}\mathsf{Rules}_{\mathbf{T1r}} := \mathsf{Rules}_{\mathbf{T1r}} \cup \{\mathsf{Rule[DetailReflexive]}, \mathsf{Rule[DetailAntisymmetric]}, \\ \mathsf{Rule[DetailTransitive]}\}\end{aligned}$$

where:

$$\begin{aligned}
\mathsf{Rule[DetailReflexive]} &\equiv \forall Z \subseteq \mathcal{L}_{\text{CPL}}\ \forall x \in \mathcal{L}_{\text{CPL}}\ \mathsf{geqDetail}(x, x, Z) \\
\mathsf{Rule[DetailAntisymmetric]} &\equiv \forall Z \subseteq \mathcal{L}_{\text{CPL}}\ \forall \{x, y\} \subseteq \mathcal{L}_{\text{CPL}}\ \mathsf{geqDetail}(x, y, Z) \\
&\quad \wedge \mathsf{geqDetail}(y, x, Z) \Leftrightarrow \mathsf{eqDetail}(x, y, Z) \\
\mathsf{Rule[DetailTransitive]} &\equiv \forall Z \subseteq \mathcal{L}_{\text{CPL}}\ \forall \{x, y, w\} \subseteq \mathcal{L}_{\text{CPL}}\ \mathsf{geqDetail}(x, y, Z) \\
&\quad \wedge \mathsf{geqDetail}(y, w, Z) \Rightarrow \mathsf{geqDetail}(x, w, Z)
\end{aligned}$$

In Rule[DetailAntisymmetric], eqDetail(x, y, Z) reads "x and y are at the same level of detail in Z".

In the refinement relation, the refining requirements must be strictly more detailed than the refined requirement. We therefore define the following variant of evaluation in terms of detail.

$$\mathsf{Rules}_{\mathbf{T1r}} := \mathsf{Rules}_{\mathbf{T1r}} \cup \{\mathsf{Rule[gDetail]}\}$$

where:

- gDetail(x, y, Z) reads "x has strictly higher level of detail than y in Z":

$$\begin{aligned}
\mathsf{Rule[gDetail]} \equiv \forall x, y \in \mathcal{L}_{\text{CPL}}\ Z \subseteq \mathcal{L}_{\text{CPL}}\ &\mathsf{geqDetail}(x, y, Z) \wedge \neg \mathsf{geqDetail}(y, x, Z) \\
&\Leftrightarrow \mathsf{gDetail}(x, y, Z)
\end{aligned}$$

To define the refinement relation, we combine the build relation and the detail evaluation. Refinement is thereby a specialization of the build relation. Moreover, we can define a taxonomy of positive relations by restricting sorts of the premises and of the conclusion in the refinement relation. The taxonomy is in Eqs. 1–5.

Eq. 1 defines a refinement relation which is independent from the sorts of premises and conclusion. Eq. 2 defines a goal refinement relation analogous to Darimont & van Lamsweerde's goal refinement [14]. Eq. 3 defines the realization, or operationalization relation, in which all premises must be tasks or assumptions. Eq. 4-5 defines the task decomposition and means-ends relations, analogous to the i-star task decomposition and means-ends relations [49].

$$\mathsf{Refine}(Y, x, Z) \Leftrightarrow \mathsf{Build}(Y, x, Z) \wedge \forall y \in Y\ \mathsf{gDetail}(y, x, Z) \tag{1}$$

$$\mathsf{GoalRefine}(Y, x, Z) \Leftrightarrow \mathsf{Refine}(Y, x, Z) \wedge \forall y \in \{x\} \cup Y\ \mathsf{G}(y, Z) \tag{2}$$

$$\mathsf{Realize}(Y, x, Z) \Leftrightarrow \mathsf{Refine}(Y, x, Z) \wedge \forall y \in Y\ (\mathsf{T}(y, Z) \vee \mathsf{A}(y, Z)) \tag{3}$$

$$\mathsf{TaskDecompose}(Y, x, Z) \Leftrightarrow \mathsf{Refine}(Y, x, Z) \wedge \mathsf{T}(x, Z) \wedge \forall y \in Y\ (\mathsf{G}(y, Z) \vee \mathsf{T}(y, Z)) \tag{4}$$

$$\mathsf{MeansEnds}(Y, x, Z) \Leftrightarrow \mathsf{Refine}(Y, x, Z) \wedge \mathsf{G}(x, Z) \wedge \forall y \in Y\ \mathsf{T}(y, Z) \tag{5}$$

If we add each equation above as a rule to **T1r**, then we can represent these relations in its Problem-Solving KB.

6.7 Evaluation Expansion

Problem-Solving evaluation expansion refers to the effort of adding new evaluation relations to a TF specification.

We illustrate evaluation expansion by adding many evaluations to **T1**. In the process, we make a new RML which we refer to as **T1me**.

Each new evaluation we add to **T1me** compares Goals, Tasks, and/or Assumptions over a single criterion for comparison. Each criterion is referred to by a constant. Let $\mathcal{E}$ be that set of constants, so that firstly we change the Domain language in **T1me** as follows.

$$\mathcal{L}_{\mathbf{D}} := \mathcal{L}_{\mathbf{T1me}}$$
$$\mathcal{L}_{\mathbf{T1me}} = \mathcal{L}_{\text{CPL}} \cup \mathcal{E}$$

where $\mathcal{E}$ is a set of constants, each referring to the name of a comparison criterion, such as cost, secutity, scalability, reliability, etc.

We use the constants from $\mathcal{E}$ in the four-place predicate $\mathsf{geqCritComp}(c, x, y, Z)$, which reads "in Z, x is at least as high as y on the scale defined by criterion c", with $c \in \mathcal{E}$, while x and y are, each, a goal, a task, or an assumption. We add this predicate to Evaluations.

$$\begin{aligned}\mathsf{Evaluations}_{\mathbf{T1me}} := \mathsf{Evaluations}_{\mathbf{T1me}} \\ \cup \{\mathsf{geqCritComp} : \mathcal{E} \times \mathsf{atoms}(\mathcal{L}_{\mathbf{T1me}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{T1me}}) \times \wp(\mathcal{L}_{\mathbf{T1me}}) \\ \longrightarrow \{\text{True}, \text{False}\}\}\end{aligned}$$

where $\mathsf{geqCritComp}(c, x, y, Z) = \text{True}$ reads "in Z, x is at least as high as y on the scale defined by criterion c", "in Z, x is not at least as high as y on the scale defined by criterion c" otherwise.

We add rules to make $\mathsf{geqCritComp}(\cdot, \cdot, \cdot, \cdot)$ a partial order relation, and to define its strict variant.

$$\begin{aligned}\mathsf{Rules}_{\mathbf{T1me}} := \mathsf{Rules}_{\mathbf{T1me}} \cup \{&\mathsf{Rule[CritCompReflexive]}, \mathsf{Rule[CritCompAntisymmetric]}, \\ &\mathsf{Rule[CritCompTransitive]}, \mathsf{Rule[gCritComp]}\}\end{aligned}$$

where:

- For every criterion, $\mathsf{geqCritComp}(\cdot, \cdot, \cdot, \cdot)$ is reflexive:

$$\mathsf{Rule[CritCompReflexive]} \equiv \forall c \in \mathcal{E}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}\ \forall x \in \mathcal{L}_{\mathbf{T1me}}\ \mathsf{geqCritComp}(c, x, x, Z)$$

- For every criterion, $\mathsf{geqCritComp}(\cdot, \cdot, \cdot, \cdot)$ is antisymmetric. $\mathsf{eqCritComp}(c, x, y, Z)$ reads "in Z, x is at same position as y on the scale defined by criterion c".

$$\begin{aligned}\mathsf{Rule[CritCompAntisymmetric]} \equiv \forall c \in \mathcal{E}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}\ \forall \{x, y\} \subseteq \mathcal{L}_{\mathbf{T1me}} \\ \mathsf{geqCritComp}(c, x, y, Z) \wedge \mathsf{geqCritComp}(c, y, x, Z) \\ \Leftrightarrow \mathsf{eqCritComp}(c, x, y, Z)\end{aligned}$$

- For every criterion, $\mathsf{geqCritComp}(\cdot, \cdot, \cdot, \cdot)$ is transitive:

$$\begin{aligned}\mathsf{Rule[CritCompTransitive]} \equiv \forall c \in \mathcal{E}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}\ \forall \{x, y, w\} \subseteq \mathcal{L}_{\mathbf{T1me}} \\ \mathsf{geqCritComp}(c, x, y, Z) \wedge \mathsf{geqCritComp}(c, y, w, Z) \\ \Rightarrow \mathsf{geqCritComp}(c, x, w, Z)\end{aligned}$$

- $\mathsf{gCritComp}(c, x, y, Z)$ reads "in Z, x is at a higher position than y on the scale defined by criterion c":

$$\begin{aligned}\mathsf{Rule[gCritComp]} \equiv \forall c \in \mathcal{E}\ \forall x, y \in \mathcal{L}_{\mathbf{T1me}}\ Z \subseteq \mathcal{L}_{\mathbf{T1me}}\\ \mathsf{geqCritComp}(c, x, y, Z) \wedge \neg\mathsf{geqCritComp}(c, y, x, Z)\\ \Leftrightarrow \mathsf{gCritComp}(c, x, y, Z)\end{aligned}$$

The new evaluations will allow us to represent nonfunctional requirements, also called softgoals or quality requirements. Examples of such requirements are "low cost" or "high security". To see how, let there be two goals x and y. Suppose that they are comparable in terms of cost, in the sense that it costs more to achieve x than to achieve y. The nonfunctional requirement "low cost" can consequently be interpreted for a given pair of goals, tasks, assumptions comparable in terms of cost, as preference for the one associated with lower cost.

We observe that a nonfunctional requirement refers to a criterion and to a direction over the scale of that criterion. We therefore add a relation to apply over criteria only, and the associated rules.

$$\begin{aligned}\mathsf{Relations}_{\mathbf{T1me}} &:= \mathsf{Relations}_{\mathbf{T1me}} \cup \{\mathsf{Increase} : \mathcal{E} \longrightarrow \{\mathrm{True}, \mathrm{False}\}\}\\ \mathsf{Rules}_{\mathbf{T1me}} &:= \mathsf{Rules}_{\mathbf{T1me}} \cup \{\mathsf{Rule[IncreaseNotDecrease]}, \mathsf{Rule[IncreaseXORDecrease]}\}\end{aligned}$$

where:

- $\mathsf{Increase}(c) = \mathrm{True}$ reads "on the scale of the criterion c, higher values are strictly preferred over lower values", "on the scale of the criterion c, higher values are not strictly preferred over lower values" otherwise.
- Increase and decrease directions are opposites, for every criterion:

$$\mathsf{Rule[IncreaseNotDecrease]} \equiv \forall c \in \mathcal{E}\ \neg\mathsf{Increase}(c) \Leftrightarrow \mathsf{Decrease}(c)$$

- We can choose one direction over every criterion, not both:

$$\begin{aligned}\mathsf{Rule[IncreaseXORDecrease]} \equiv \forall c \in \mathcal{E}\ (\mathsf{Increase}(c) \vee \mathsf{Decrease}(c))\\ \wedge \neg(\mathsf{Increase}(c) \wedge \mathsf{Decrease}(c))\end{aligned}$$

If $dcost \in \mathcal{E}$ and $dcost$ informally reads "development cost", then $\mathsf{Decrease}(dcost)$ captures the nonfunctional requirement "low development cost". We can use these softgoals to generate preferences over comparisons, as follows. We add a criterion-specific preference relation, $\mathsf{geqPreference}(\cdot, \cdot, \cdot, \cdot)$, such that $\mathsf{geqPreference}(c, x, y, Z)$ reads that for the criterion c and in Z, x is at least as desirable as y. The preference relation over each criterion is a partial order.

$$\begin{aligned}\mathsf{Evaluations}_{\mathbf{T1me}} &:= \mathsf{Evaluations}_{\mathbf{T1me}} \\ &\quad \cup \{\mathsf{geqPreference} : \mathcal{E} \times \mathsf{atoms}(\mathcal{L}_{\mathbf{T1me}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{T1me}}) \times \wp(\mathcal{L}_{\mathbf{T1me}}) \\ &\quad \longrightarrow \{\mathrm{True}, \mathrm{False}\}\} \\ \mathsf{Rules}_{\mathbf{T1me}} &:= \mathsf{Rules}_{\mathbf{T1me}} \\ &\quad \cup \{\mathsf{Rule[PreferenceReflexive]}, \mathsf{Rule[PreferenceAntisymmetric]}, \\ &\quad \mathsf{Rule[PreferenceTransitive]}, \mathsf{Rule[gPreference]}\}\end{aligned}$$

where:

- $\mathsf{geqPreference}(c, x, y, Z) = \mathrm{True}$ reads "in Z, and for the criterion c, x is at least as desirable as y", "in Z, and for the criterion c, x is not at least as desirable as y" otherwise.
- For every criterion, $\mathsf{geqPreference}(\cdot, \cdot, \cdot, \cdot)$ is reflexive:

$$\mathsf{Rule[PreferenceReflexive]} \equiv \forall c \in \mathcal{E}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}\ \forall x \in \mathcal{L}_{\mathbf{T1me}}\ \mathsf{geqPreference}(c, x, x, Z)$$

- For every criterion, $\mathsf{geqPreference}(\cdot, \cdot, \cdot, \cdot)$ is antisymmetric. $\mathsf{eqPreference}(c, x, y, Z)$ reads "in Z, and for the criterion c, x is as desirable as y".

$$\begin{aligned}\mathsf{Rule[PreferenceAntisymmetric]} \equiv \forall c \in \mathcal{E}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}\ \forall \{x, y\} \subseteq \mathcal{L}_{\mathbf{T1me}} \\ \mathsf{geqPreference}(c, x, y, Z) \wedge \mathsf{geqPreference}(c, y, x, Z) \\ \Leftrightarrow \mathsf{eqPreference}(c, x, y, Z)\end{aligned}$$

- For every criterion, $\mathsf{geqPreference}(\cdot, \cdot, \cdot, \cdot)$ is transitive:

$$\begin{aligned}\mathsf{Rule[PreferenceTransitive]} \equiv \forall c \in \mathcal{E}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}\ \forall \{x, y, w\} \subseteq \mathcal{L}_{\mathbf{T1me}} \\ \mathsf{geqPreference}(c, x, y, Z) \wedge \mathsf{geqPreference}(c, y, w, Z) \\ \Rightarrow \mathsf{geqPreference}(c, x, w, Z)\end{aligned}$$

- $\mathsf{gPreference}(c, x, y, Z)$ reads "in Z, and for the criterion c, x is strictly more desirable than y":

$$\begin{aligned}\mathsf{Rule[gPreference]} \equiv \forall c \in \mathcal{E}\ \forall x, y \in \mathcal{L}_{\mathbf{T1me}}\ Z \subseteq \mathcal{L}_{\mathbf{T1me}} \\ \mathsf{geqCritComp}(c, x, y, Z) \wedge \neg\mathsf{geqPreference}(c, y, x, Z) \\ \Leftrightarrow \mathsf{gPreference}(c, x, y, Z)\end{aligned}$$

Given a softgoal, we can generate preference relations: if we have $\mathsf{Increase}(c)$ for the criterion c, and we have a comparison of x and y as follows $\mathsf{gCritComp}(c, x, y, Z)$, then we also know that we prefer x to y, i.e., that $\mathsf{gPreference}(c, x, y, Z)$. The following rules generate these strict preferences from nonfunctional requirements and comparisons.

$$\mathsf{Rules}_{\mathbf{T1me}} := \mathsf{Rules}_{\mathbf{T1me}} \cup \{\mathsf{Rule[IncreasePref]}, \mathsf{Rule[DecreasePref]}\}$$

where:

- If there is $\mathsf{Increase}(c)$ and $\mathsf{gCritComp}(c, x, y, Z)$, then there is also $\mathsf{gPreference}(c, x, y, Z)$:

$$\mathsf{Rule[IncreasePref]} \equiv \forall c \in \mathcal{E}\ \forall x, y \in \mathcal{L}_{\mathbf{T1me}}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}$$
$$\mathsf{Increase}(c) \wedge \mathsf{gCritComp}(c, x, y, Z) \Rightarrow \mathsf{gPreference}(c, x, y, Z)$$

- If there is $\mathsf{Decrease}(c)$ and $\mathsf{gCritComp}(c, x, y, Z)$, then there is also $\mathsf{gPreference}(c, y, x, Z)$:

$$\mathsf{Rule[DecreasePref]} \equiv \forall c \in \mathcal{E}\ \forall x, y \in \mathcal{L}_{\mathbf{T1me}}\ \forall Z \subseteq \mathcal{L}_{\mathbf{T1me}}$$
$$\mathsf{Decrease}(c) \wedge \mathsf{gCritComp}(c, x, y, Z) \Rightarrow \mathsf{gPreference}(c, y, x, Z)$$

6.8 Strengthening

Given two TF specifications A and B, we say that A is stronger than B if the only difference between them is that A includes more rules than B. That is, if $\mathsf{Rules}_B \subset \mathsf{Rules}_A$ and everything else in their TF specifications is the same.

Problem-Solving strenghtening consists of changing an original TF specification only by adding new Rules, so that the strenghtened specification does not exibit some drawbacks of the original.

For illustration, consider what happens with **T1** if it is applied to an inconsistent Domain KB X, i.e., $X \vdash \bot$.

The consequence relation $\vdash$ satisfies *ex falso quodlibet*, so that anything can be concluded from an inconsistent set of expressions in classical propositional logic. There will consequently be at least as many Build relations as there are expressions in $\mathcal{L}_{\text{CPL}}$. There will be at least one Build relation to any goal atom.

Suppose that we made **T0** by taking **T1** and making two changes: (i) we remove Rule[BreakFree], and (ii) if a rule mentions the predicate $\mathsf{BreakFree}(\cdot, \cdot)$, we change the rule so it no longer mentions it. So **T0** can still represent Break relations, but these relations would have no influence on sort statuses. This would ensure that **T0** would mark as achieved all goals in any inconsistent Domain KB. Achievement would be due to inconsistency, not to knowledge relevant to the achievement of the goal.

T1 is a strenghening of **T0** in the sense that the only difference between **T0** and **T1** are the rules, and **T1** will not fail in the way that **T0** does, when Domain KB is inconsistent.

7 RML Merging

RML merging consists of combining TF specifications of two or more existing RMLs, in order to create the specification of a new RML. We illustrate Problem-Solving merging by combining **T1** and **iS1** into a new TF specification, and we refer to the resulting RML as **TS1**. Our aim in merging **T1** and **iS1** is that the new specification should allow us to say at least everything we could with each of them separately.

In order to say with the Domain KB of **TS1** at least the same as with **iS1** and **T1** separately, the merged Domain language is that of CPL, to which we add a set of constants. The consequence relation is the consequence relation $\vdash$ of CPL; as it is reflexive, all atoms in a Domain KB are also in the closure of that KB, which fits our definition of $\mid\!\sim_*$ in **iS1**. While we add constants to the language of CPL, we keep the original $\vdash$ of CPL, so that constants are independent from the inference rules; this follows $\mid\!\sim_*$ from **iS1**.

$\mathsf{D}_{\mathbf{TS1}} = (\mathcal{L}_{\mathbf{TS1}}, \vdash, X)$, where:

- $\mathcal{L}_{\mathbf{TS1}} = \mathcal{L}_{\text{CPL}} \cup \mathcal{C}$, where $\mathcal{L}_{\text{CPL}}$ is the language of classical propositional logic and $\mathcal{C}$ is a set of constants. Constants can be, for example, names of people, names of roles and positions, names of resources, etc.
- $\vdash$ is the consequence relation of classical propositional logic.
- X is some subset of $\mathcal{L}_{\mathbf{TS1}}$.

The merged Interface Knowledge is simply a union of selectors from **T1** and **iS1**.

$$\mathsf{I}_{\mathbf{TS1}} = \{\mathsf{atoms}(\cdot), \mathsf{minproofs}(\cdot), \mathsf{premises}(\cdot,\cdot), \mathsf{conclusion}(\cdot,\cdot), \mathsf{constants}(\cdot)\}$$

The merged Problem-Solving Knowledge has the same Problem-Solving language, satisfiability relation, and KB as in **T1** and **iS1**. We discuss the rest of Problem-Solving Knowledge below.

It is useful to summarize key differences between the Problem-Solving Knowledge of **T1** and of **iS1**, in order to understand how we will merge their Sorts, Statuses, Relations, and Rules:

1. In **T1**, all predicates in Sorts, Statuses, and Relations include a term which is always a set of Domain expressions. For example, in $\mathsf{Goal}(x, Y)$ this set is the second term. The purpose of that set is to make sorting, status assignments, and relations local to a set of Domain expressions, and allow different sortings and status assignments to same Domain expressions, when they are in other sets. This can be useful when the Domain KB is inconsistent, and we want to look at, for example, only its maximally consistent subsets, and perhaps the effects on the presence or absence of solutions in these sets as a function of sorts we assign to atoms in them. There was no need for this in **iS1**, because there was no relation analogous to Break in **T1**, so that it was impossible to say that, for example, two goals cannot be achieved together.

2. The key idea in **iS1** was that of dependency relations, to capture that some agents want to delegate to other agents the achievement of goals, the execution of tasks, and the delivery of resources. There were no agents in **T1** – requirements were independent of who owns them, and who is resonsible for making them succeed.

As **TS1** will include the break relation, we will change all sort, status, and relation predicates carried over from **iS1** so that they are local to a set of Domain expressions. To be able to capture dependency relations in **Tts1**, we will carry over all sorts over constants from **iS1**. We keep the goal and task sorts from **T1**, and do not carry over those from **iS1**.

$$\begin{aligned}\mathsf{Sorts}_{\mathbf{TS1}} := \ \{&\mathsf{G}(\cdot,\cdot), \mathsf{T}(\cdot,\cdot), \mathsf{A}(\cdot,\cdot), \mathsf{Resource}(\cdot,\cdot),\\ &\mathsf{Agent}(\cdot,\cdot), \mathsf{Role}(\cdot,\cdot), \mathsf{PositionName}(\cdot,\cdot)\}\end{aligned}$$

We omit the definitions of predicates above, and in the rest of this section, as they are straightforward adaptations of definitions given earlier for **T1** and **iS1**. Notice that predicates are now over expressions and constants from the **TS1** language.

The merged Statuses includes all those from **T1**, and the updated status for resources, which is now local to a set of Domain expressions.

$$\mathsf{Statuses}_{\mathbf{TS1}} := \{\mathsf{Achieved}(\cdot,\cdot), \mathsf{Maintained}(\cdot,\cdot), \mathsf{Executed}(\cdot,\cdot), \mathsf{Delivered}(\cdot,\cdot)\}$$

In **iS1**, relations are never between goals, tasks, and resources. More generally, an atom is never related to another atom. In contrast, there are no constants in **T1** and all relations are between atoms. We keep the build and break relations from **T1**, and all relations from **iS1**. But we also change each relation from **iS1** by indicating, as we did for sorts, that it is local to a set of Domain expressions. As we are not changing the definitions of the build and break relations, only expressions can be in these relations, and therefore not constants; there can be no break or build relation between agents, roles, positions, and resources.

As we now have assumptions as the fourth sort next to goal, task, and resource, we need to be able to define dependencies over assumptions. For this reason, we add $\mathsf{WantsMaintained}(\cdot,\cdot,\cdot)$ and $\mathsf{CanMaintain}(\cdot,\cdot,\cdot)$ relations.

$$\begin{aligned}\mathsf{Relations}_{\mathbf{TS1}} := \{&\mathsf{Build}(\cdot,\cdot,\cdot), \mathsf{Break}(\cdot,\cdot), \mathsf{Plays}(\cdot,\cdot,\cdot),\\ &\mathsf{WantsAchieved}(\cdot,\cdot,\cdot), \mathsf{CanAchieve}(\cdot,\cdot,\cdot),\\ &\mathsf{WantsExecuted}(\cdot,\cdot,\cdot), \mathsf{CanExecute}(\cdot,\cdot,\cdot),\\ &\mathsf{WantsDelivered}(\cdot,\cdot,\cdot), \mathsf{CanDeliver}(\cdot,\cdot,\cdot),\\ &\mathsf{WantsMaintained} : \mathsf{constants}(\mathcal{L}_{\mathbf{TS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{TS1}}) \times \wp(\mathcal{L}_{\mathbf{TS1}} \setminus \mathcal{C})\\ &\longrightarrow \{\mathrm{True}, \mathrm{False}\},\\ &\mathsf{CanMaintain} : \mathsf{constants}(\mathcal{L}_{\mathbf{TS1}}) \times \mathsf{atoms}(\mathcal{L}_{\mathbf{TS1}}) \times \wp(\mathcal{L}_{\mathbf{TS1}} \setminus \mathcal{C})\\ &\longrightarrow \{\mathrm{True}, \mathrm{False}\}\}\end{aligned}$$

where:

- $\mathsf{WantsMaintained}(a, x, Y) = \mathrm{True}$ reads "in Y, agent a wants assumption x maintained", "in Y, agent a does not want assumption x maintained" otherwise.
- $\mathsf{CanMaintain}(a, x, Y) = \mathrm{True}$ reads "in Y, agent a can maintain assumption x", "in Y, agent a cannot maintain assumption x" otherwise.

Since $\mathsf{Resource}(\cdot,\cdot)$ classifies constants, and not atoms, as resources, we adopt the Rule[GAT] from **T1**. We also keep all rules that do *not* define conditions for the success or failure of a goal, assumption, or task. This is because we want to combine these conditions from **T1** with those in **iS1**; roughly, this means that the achievement of a goal, for example, will depend both on their being good builds to it, it being break free, and on it being in a goal dependency; the former conditions were missing in **iS1**, while the latter was missing in **T1**.

$$\begin{aligned}\mathsf{Rules}_{\mathbf{TS1}} := \{&\mathsf{Rule[GAT]}, \mathsf{Rule[Build]}, \mathsf{Rule[Break]}, \mathsf{Rule[Primitive]},\\ &\mathsf{Rule[BreakFree]}, \mathsf{Rule[GoodBuild]}\}\end{aligned}$$

We carry over all rules for dependency relations from **iS1**, and we add the assumption dependency.

$$\mathsf{Rules}_{\mathbf{TS1}} := \mathsf{Rules}_{\mathbf{TS1}} \cup \{\mathsf{Rule[GoalDependency]}, \mathsf{Rule[TaskDependency]}, \mathsf{Rule[ResourceDependency]}, \mathsf{Rule[AssumptionDependency]}\}$$

where:

- That agent a depends on agent b to have assumption x maintained means that a needs to have x maintained, cannot do it herself, and b is able to maintain x:

$$\begin{aligned}\mathsf{Rule[AssumptionDependency]} \equiv \exists a, b, x, Y\ & \mathsf{Assumption}(x, Y) \wedge \mathsf{Agent}(a, Y) \\ & \wedge \mathsf{WantsMaintained}(a, x, Y) \wedge \neg\mathsf{CanMaintain}(a, x, Y) \\ & \wedge \mathsf{Agent}(b, Y) \wedge \mathsf{CanMaintain}(b, x, Y) \\ & \Leftrightarrow \mathsf{AssumptionDependency}(a, b, x, Y)\end{aligned}$$

To define achievement, execution, delivery, and maintenance, we adopt Rule[Dependum] from **iS1** and redefine it as follows.

$$\mathsf{Rules}_{\mathbf{TS1}} := \mathsf{Rules}_{\mathbf{TS1}} \cup \{\mathsf{Rule[Dependum]}\}$$

where:

- Any goal, task, resource, or assumption x is a dependum if there is a dependency relation on it:

$$\begin{aligned}\mathsf{Rule[Dependum]} \equiv \forall x\ (\exists a, b, Y\ & \mathsf{GoalDependency}(a, b, x, Y) \\ & \vee \mathsf{TaskDependency}(a, b, x, Y) \vee \mathsf{ResourceDependency}(a, b, x, Y) \\ & \vee \mathsf{AssumptionDependency}(a, b, x, Y)) \\ & \Leftrightarrow \mathsf{Dependum}(x, Y)\end{aligned}$$

We can now define the conditions for goal achievement, task execution, resource delivery, and assumption maintenance.

$$\mathsf{Rules}_{\mathbf{TS1}} := \mathsf{Rules}_{\mathbf{TS1}} \cup \{\mathsf{Rule[Achieved]}, \mathsf{Rule[Executed]}, \mathsf{Rule[Delivered]}, \mathsf{Rule[Maintained]}, \mathsf{Rule[Successful]}, \mathsf{Rule[Failed]}\}$$

where:

- A goal x is achieved in the set Y if (i) there is an agent who wants it achieved, (ii) another agent who can achieve it, (iii) x is not in Break relations, and (iv) there is a good Build relation to x, in which all premises are successful:

$$\begin{aligned}\mathsf{Rule[Achieved]} \equiv \forall Y, Z \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}}\ & \\ & \mathsf{G}(x, Y) \wedge \mathsf{Dependum}(x, Y) \wedge \mathsf{BreakFree}(x, Y) \\ & \wedge \mathsf{GoodBuild}(x, Z, Y) \wedge \forall w \in Z\ \mathsf{Succeeds}(w, Y)) \\ & \Leftrightarrow \mathsf{Achieved}(x, Y)\end{aligned}$$

- An assumption x is maintained in the set Y if (i) there is an agent who wants it maintained, (ii) another agent who can maintain it, (iii) x is in no Break relations, and either (iv-a) is primitive, or, if it is not primitive, (iv-b) then there needs to be a good Build to it, in which all premises are successful:

$$\begin{aligned}\mathsf{Rule[Maintained]} \equiv \forall Y, Z \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}}\\ \mathsf{A}(x,Y) \wedge \mathsf{Dependum}(x,Y) \wedge \mathsf{BreakFree}(x,Y)\\ \wedge (\mathsf{Primitive}(x,Y) \vee (\neg\mathsf{Primitive}(x,Y)\\ \wedge \mathsf{GoodBuild}(x,Z,Y) \wedge \forall w \in Z\ \mathsf{Succeeds}(w,Y)))\\ \Leftrightarrow \mathsf{Maintained}(x,Y)\end{aligned}$$

- A task x is executed in the set Y if (i) there is an agent who wants it executed, (ii) another agent who can execute it, (iii) x is in no Break relations, and either (iv-a) primitive, or, if it is not primitive, (iv-b) then there needs to be a good Build to it, in which all premises are successful:

$$\begin{aligned}\mathsf{Rule[Executed]} \equiv \forall Y, Z \subseteq \mathcal{L}_{\text{CPL}}\ x \in \mathcal{L}_{\text{CPL}}\\ \mathsf{T}(x,Y) \wedge \mathsf{Dependum}(x,Y) \wedge \mathsf{BreakFree}(x,Y)\\ \wedge (\mathsf{Primitive}(x,Y) \vee (\neg\mathsf{Primitive}(x,Y)\\ \wedge \mathsf{GoodBuild}(x,Z,Y) \wedge \forall w \in Z\ \mathsf{Succeeds}(w,Y)))\\ \Leftrightarrow \mathsf{Executed}(x,Y)\end{aligned}$$

- A resource is delivered in the set Y if there is an agent who wants it delivered and another who can deliver it:

$$\begin{aligned}\mathsf{Rule[Delivered]} \equiv \forall x \in \mathcal{L}_{\mathbf{TS1}} \setminus \mathcal{L}_{\text{CPL}}\ \mathsf{Resource}(x,Y) \wedge \mathsf{Dependum}(x,Y)\\ \Leftrightarrow \mathsf{Delivered}(x,Y)\end{aligned}$$

- Success is defined as goal achievement, assumption maintenance, task execution, or resource delivery:

$$\begin{aligned}\mathsf{Rule[Successful]} \equiv \forall x, Y\ (\mathsf{Achieved}(x,Y) \vee \mathsf{Maintained}(x,Y) \vee \mathsf{Executed}(x,Y)\\ \vee \mathsf{Delivered}(x,Y)) \Leftrightarrow \mathsf{Succeeds}(x,Y)\\ \mathsf{Rule[Failed]} \equiv \forall x, Y\ (\neg\mathsf{Achieved}(x,Y) \vee \neg\mathsf{Maintained}(x,Y) \vee \neg\mathsf{Executed}(x,Y)\\ \vee \mathsf{Delivered}(x,Y)) \Leftrightarrow \neg\mathsf{Succeeds}(x,Y)\end{aligned}$$

As there are break relations in **TS1**, we cannot ask that all goals, tasks, assumptions, and resources be successful. Instead, we let the problem and solution concepts be those from **T1**, i.e., a solution makes all goals successful.

There were no evaluations in **T1** and in **iS1**, and there are none in **TS1**. If needed, one can add evaluations to **TS1** in the same way as described for evaluation introduction in Section 6.6.

8 RML Analysis

Given a TF specification of an RML, we want to evaluate if it satisfies some properties of interest. We mentioned these properties in the introduction, and we discuss each of them in this section. This is an incomplete list, other properties can be relevant.

8.1 Conciseness

Given two TF specifications **A** and **B**, where **B** was obtained only by removing parts from **A**, we say that **A** fails Conciseness if solution instances of **B** are also solution instances of **A**.

To illustrate Conciseness, we use **T1r** from Section 6.6. **T1r** is **T1** to which we added the evaluation relation $\mathsf{geqDetail}(\cdot,\cdot,\cdot)$ to represent comparisons in terms of level of detail. We showed how to define its strict variant, $\mathsf{gDetail}(\cdot,\cdot,\cdot)$, and we used it to define refinement and a taxonomy of refinement relations.

Suppose that we now make another RMLSpec, we denote it **T1rx**, and we define it as follows: **T1rx** := **T1r** and we add to it the following rules:

$$\mathsf{Rules}_{\mathbf{T1rx}} := \mathsf{Rules}_{\mathbf{T1rx}} \cup \{\mathsf{Rule[Refinement]}, \mathsf{Rule[GoalRefinement]}, \mathsf{Rule[Realization]}\}$$

where:

$$\begin{aligned}
\mathsf{Rule[Refinement]} \equiv\ & \forall x, Y, Z\ \mathsf{Refine}(Y, x, Z) \\
& \Leftrightarrow \mathsf{Build}(Y, x, Z) \wedge \forall y \in Y\ \mathsf{gDetail}(y, x, Z) \\
\mathsf{Rule[GoalRefinement]} \equiv\ & \forall x, Y, Z\ \mathsf{GoalRefine}(Y, x, Z) \\
& \Leftrightarrow\ \mathsf{Refine}(Y, x, Z) \wedge \forall y \in \{x\} \cup Y\ \mathsf{G}(y, Z) \\
\mathsf{Rule[Realization]} \equiv\ & \forall x, Y, Z\ \mathsf{Realize}(Y, x, Z) \\
& \Leftrightarrow\ \mathsf{Refine}(Y, x, Z) \wedge \forall y \in Y\ (\mathsf{T}(y, Z) \vee \mathsf{A}(y, Z))
\end{aligned}$$

Removing these rules from **T1rx** makes no difference to the RML that the specification represents. We defined the additional rules above using ingredients already in **T1r**. **T1rx** fails Conciseness. Notice that Conciseness fails for **T1rx** because the conditions for the success of goals, tasks, and assumptions in **T1rx** are unrelated to there being refinements, goal refinements, or realization relations in the Domain KB.

Conciseness is not necessarily a desirable property. For example, it is independent from the ease of use of Problem-Solving Knowledge that is specified. It may be the case that **T1rx** is more convenient than **T1r**. : it may be helpful, *while searching for solution instances*, to have the refinement, goal refinement, and realization relations. But if we choose that th problem and solution concepts be independent from these relations, the specification will fail Conciseness.

8.2 Clarity

A TF specification succeeds Clarity if we can always determine whether we found zero or more solution instances when using the RML it specifies.

The most apparent case when Clarity fails is when a TF specification is missing either or both the problem and the solution predicates, that is, when they are undefined. If they are defined, it will fail if it is unclear from the rules when the sorts have desirable, and when they have undesirable statuses, where desirable statuses are those required in the solution, and undesirable those required in the problem.

For illustration, recall that when we defined **iS1** from **i**$*$, we introduced statuses, the problem, and the solution concept not on the basis of the publication we selected. Because there was no clear problem statement in in the publication we selected as our primary source of knowledge on **i**$*$, and there was no notion of solution, we needed to decide how an **i**$*$ model needs to look like, in order for it to be problematic, or for it to include a solution instance.

If we take **iS1** and remove the problem and solution concepts from it, we obtain a TF specification which fails Clarity.

Whether Clarity is a desirable property depends on the RML being specified. **i∗** was introduced as a modeling language for a wide range of application areas. As different application areas may have their own specific problems that **i∗** can be used to solve, a TF specification of **i∗** *without* the problem and solution predicates can be taken as a template, waiting to be adapted for specific needs within each application area.

8.3 Decisiveness

A TF specification fails Decisiveness if we can find more than one solution instance for the same problem instance.

In **T1**, the conditions for a set to be a solution instance do not restrict the possibility that there are several such sets in the Domain KB, so that **T1** failed Decisiveness. **iS1**, **iS2**, and **iS3** all fail Decisiveness whenever all goals, tasks, and assumptions in the Domain KB are in at least one dependency relation, and at least one goal, task, or assumption is in more than one dependency relation. In that case, there is nothing in either **iS1**, **iS2**, or **iS3** that tells us which dependency relations we should keep, and which we can remove from a solution.

When there are several solutions we can choose from, the usual approach is to define criteria, compare solutions over the criteria, establish a total order of solutions based on how they rank over all criteria, and select one which is not dominated in that total order. We introduced criteria in **T1me**, but did not introduce rules for how to use the criteria to establish a total order of all solutions.

We can make sure in different ways that **T1me** satisfies Decisiveness. A simple option is to focus on one criterion only, and require that the solution be one set in which none of the members ranks lower on that criterion than in another set, and that, in case there are several sets equivalent on that criterion, one of them be chosen randomly. Problem remains the same, and solution concept is replaced as follows, when the chosen criterion is denoted c.

$$
\begin{aligned}
&(\exists X \subseteq \mathcal{L}_{\text{CPL}}\ \neg\mathsf{Problem}(X) \wedge (\exists z \in X\ \mathsf{G}(z, X)) \\
&\wedge \exists Y \subseteq X\ \forall z \in X\ (\mathsf{G}(z, X) \Rightarrow (\mathsf{G}(z, Y) \wedge \mathsf{Succeed}(z, Y))) \\
&(\nexists W \subseteq X\ \forall z \in X\ (\mathsf{G}(z, X) \Rightarrow (\mathsf{G}(z, W) \wedge \mathsf{Succeed}(z, W))) \\
&\wedge \exists r \in W, q \in Y\ \mathsf{gCritComp}(c, r, q, X))\ ) \\
&\Leftrightarrow \mathsf{Solution}(Y, X)
\end{aligned}
$$

9 Related Work

TF is our current response to the problems of having to design, relate, and compare many RMLs. It is based on the assumptions that (i) there is no unique or universal statement of the problem that RE aims to solve, (ii) there is no universal RML that can be used to solve any requirements problem we may encounter when doing RE, and (iii) there is consequently a need for knowledge about how to make, compare, extend, merge, and analyze RMLs in a systematic way.

Many ideas in TF are due to individual RMLs that precede it. RMF influenced the decision not to commit to one Domain formalism, but leave that choice open to the RML designer who uses TF. RMF showed that requirements ought to capture knowledge about the domain, and that there can be a need for powerful formalisms when doing so. But perhaps such formalisms are not always necessary, and simpler ones may be good enough. Also, RMF committed to the sorts entity, activity, and assertion, which do not reflect clearly enough that we are also interested in why some requirement is present, namely, because there may be some goal that the system-to-be should help its stakeholders achieve. We and others have argued for other sorts (e.g., [13, 52, 31]), and it is a disagreement over the right

set of sorts that led us to avoid committing to specific sorts, statuses, relations, and evaluations in TF. Such decisions are left to the RML designer. KAOS and Formal Tropos [19] inspired the idea that Problem-Solving Knowledge *is about* Domain Knowledge, or in other words, that sorts and relations can be seen as a mechanism to structure Domain Knowledge: KAOS in some sense does this by organizing ground formulas of linear temporal logic into goals, and proofs into goal refinements, or conflicts between goals. LQCL and Techne led us to avoid committing to a single set of inference rules in Problem-Solving Knowledge: both are paraconsistent, but in different ways, and there are many other paraconsistent formalisms, each arguing for its own set of inference rules. This also led to the notion of statuses, since conditions for the assignment of statuses to sorted Domain expressions clarify conditions that a Problem-Solving KB should satisfy, for us to be happy with it; this means that we can distinguish the fact that some, for example, goal expression can be derived from the Domain or Problem-Solving KB, from us concluding that that goal is achieved. In other words, we do not equate the derivability or satisfiability of an expression with it being, and this depending on the sort, achieved, maintained, executed, or otherwise.

10 Conclusions

RE uses requirements models. Such models are made with RMLs. We proposed the Techne Framework for the formal specification of RMLs. We showed how to use TF (i) to specify a new RML, (ii) to specify an existing RML, (iii) to specify changes to RMLs, (iv) to specify merged RMLs, and (v) to analyze RMLs by analyzing interesting properties of their TF specifications.

There are many open questions. We did not discuss how we would specify knowledge about how to use an RML, that is, steps in which one can best use it. We saw this as an issue for the procedural specification of RMLs, and left it outside the scope of this paper. We defined only the Conciseness, Clarity, and Decisiveness properties of TF specifications. There may be other properties of interest when comparing RMLs. We did not consider if it would be feasible to avoid defining new RMLs whenever we specify an extension to an existing RML, but rather specify RML modules that can be used to extend more than one RML. We did not consider how best practices in RMLs, such as the use of specific sorts (e.g., the goal sort), of specific relations (e.g., refinement) can be specified as properties of TF specifications, so that we can check these properties on existing and new RMLs.

References

[1] Anonymous. Report of the Inquiry Into The London Ambulance Service. Technical report, The Communications Directorate, South West Thames Regional Authority, 1993.

[2] Patrik Berander and Anneliese Andrews. Requirements prioritization. In *Engineering and managing software requirements*, pages 69–94. Springer, 2005.

[3] Ph. Besnard and A. Hunter. Quasi-classical logic: Non-trivializable classical reasoning from inconsistent information. In C. Froideveaux and J. Kohlas, editors, *Symbolic and Quantitative Approaches to Uncertainty*. Springer, 1995.

[4] B. Boehm, P. Bose, E. Horowitz, and M. J. 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.

[5] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approachesa survey. *Annals of Software Engineering*, 10(1-4):177–205, 2000.

[6] Barry W Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.

[7] Barry W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE*, pages 592–605, 1976.

[8] 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.

[9] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Inf. Syst.*, 27(6):365–389, 2002.

[10] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[11] 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.

[12] Jane Cleland-Huang, Grant Zemont, and Wiktor Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 230–239. IEEE, 2004.

[13] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.

[14] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *SIGSOFT FSE*, 1996.

[15] 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.

[16] E. Dubois, J. Hagelstein, and A. Rifaut. Formal Requirements Engineering with ERAE. *Philips Journal of Research*, 43(3/4):393–414, 1988.

[17] Neil A Ernst, Alexander Borgida, John Mylopoulos, and Ivan J Jureta. Agile requirements evolution via paraconsistent reasoning. In *Advanced Information Systems Engineering*, pages 382–397. Springer, 2012.

[18] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994.

[19] A. Fuxman, L. Liu, J. Mylopoulos, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Eng.*, 9(2):132–150, 2004.

[20] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. In *Conceptual ModelingER 2002*, pages 167–181. Springer, 2003.

[21] J. A. Goguen and C. Linde. Techniques for requirements elicitation. In *Proc. Int. Symp. Req. Eng.*, pages 152–164, 1993.

[22] 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.

[23] S. J. Greenspan, J. Mylopoulos, and A. Borgida. Capturing more world knowledge in the requirements specification. In *Proc. 6th Int. Conf. Software Eng.*, pages 225–234, 1982.

[24] Mats Per Erik Heimdahl and Nancy G Leveson. Completeness and consistency in hierarchical state-based requirements. *Software Engineering, IEEE Transactions on*, 22(6):363–377, 1996.

[25] 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.

[26] 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.

[27] Ann M Hickey and Alan M Davis. A unified model of requirements elicitation. *Journal of Management Information Systems*, 20(4):65–84, 2004.

[28] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: Reasoning, analysis, and action. *ACM Trans. Softw. Eng. Methodol.*, 7(4):335–367, 1998.

[29] I. J. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos. Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling. In *18th IEEE Int. Requirements Eng. Conf.*, 2010.

[30] I J Jureta and S 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.

[31] I. J. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the core ontology and problem in requirements engineering. In *16th IEEE Int. Requirements Eng. Conf.*, pages 71–80, 2008.

[32] I. J. Jureta, J. Mylopoulos, and S. Faulkner. Analysis of multi-party agreement in requirements validation. In *17th IEEE Int. Requirements Engineering Conf.*, 2009.

[33] Haruhiko Kaiya and Motoshi Saeki. Using domain ontology as domain knowledge for requirements elicitation. In *Requirements Engineering, 14th IEEE International Conference*, pages 189–198. IEEE, 2006.

[34] Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, 39(14):939–947, 1998.

[35] 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.

[36] J. C. S. P. Leite and P. A. Freeman. Requirements validation through viewpoint resolution. *IEEE T. Softw. Eng.*, 17(12):1253–1269, 1991.

[37] E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *SIGSOFT FSE*, pages 53–62, 2004.

[38] H. J. Levesque. A logic of implicit and explicit belief. In *AAAI Proceedings*, 1984.

[39] 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.

[40] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng.*, 18(6):483–497, 1992.

[41] B. Nuseibeh and S. M. Easterbrook. Requirements engineering: a roadmap. In Anthony Finkelstein, editor, *ICSE - Future of SE Track*, pages 35–46. ACM, 2000.

[42] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, 2001.

[43] Guttorm Sindre and Andreas L Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[44] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997.

[45] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Software Eng.*, 24(11):908–926, 1998.

[46] 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.

[47] R. Wieringa, N. Maiden, N. Mead, and C. Rolland. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering*, 11(1):102–107, 2006.

[48] E. Yu. Towards modeling and reasoning support for early requirements engineering. In *Proc. 3rd IEEE Int. Symposium on Requirements Eng.*, pages 226–235, 1997.

[49] E. S. K. Yu and J. Mylopoulos. Understanding "Why" in Software Process Modelling, Analysis, and Design. In *Proc. 16th Int. Conf. Software Eng.*, pages 159–168, 1994.

[50] 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.

[51] P. Zave. Classification of research efforts in requirements engineering. *ACM Comput. Surv.*, 29(4):315–321, 1997.

[52] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM T. Softw. Eng. Methodol.*, 6(1):1–30, 1997.

[53] Didar Zowghi and Vincenzo Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993–1009, 2003.