Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

# Environment Support for Requirements Writing and Analysis

Vincenzo Gervasi

Supervisor
Prof. Vincenzo Ambriola

December 30, 1999

# Abstract

In this thesis we consider the problem of providing automatic support to a number of activities performed in the context of requirements engineering. Requirements engineering focuses on the process of building a complete and consistent description (a requirements document) of the characteristics that a software-intensive system should exhibit in order to solve the customer's problem at which it is targeted. Requirements engineering includes disparate activities like information gathering, models building and analysis, specification, quality and process control, and many others. Given that the risk factor for requirements engineering is usually quite high, due to intrinsic difficulties and to the high cost of correcting errors made in requirements, providing integrated, automatic support to as many requirements engineering activities as possible is a critical issue.

In this work we concentrate mainly on environment support for requirements writing and analysis. We define basic requirements for an environment supporting requirements engineering, and present an extensible, component-based architecture that satisfies those requirements. We also present an implementation of our architecture, together with a number of components aimed at producing document models, system models, and process models from a set of natural language requirements, at validating those models, and at measuring their characteristics. In the course of the presentation, a number of innovative techniques and tools are introduced.

The thesis is completed by two case studies, discussing applications of our environment in real cases, and evaluating the effectiveness of our approach.

To my family,
and especially to my father,
who could not read this work.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Introduction

## I.1 Subject area

Requirements engineering focuses on the process of building a complete and consistent description (a requirements document) of the characteristics that a software-intensive system should exhibit in order to solve the customer's problem at which it is targeted. Requirements engineering includes disparate activities like information gathering, models building and analysis, specification, quality and process control, and many others.

Many of these activities require interaction of many parties. For example, information gathering involves at least a requirements engineer, trying to obtain information relevant for a problem in order to state the requirements for a system solving the problem, and a customer, trying to provide all the information needed about the problem and about his or her desires regarding a solution to the problem. Communication among different parties can be hindered by differences in knowledge, experience, skills, and cultural background; this is known as the "communication gap" in the literature.

Incomplete or incorrect communication also makes it difficult to externalize both the customer's and the requirements engineer's mental models of a system. This, in turn, hampers both parties in making use of such models, e.g., for getting a better understanding of the problem, for validation purposes, or for collecting metric data about them. The problem increases in complexity when more than two parties are involved — as is the case for many real requirements processes.

## I.2 Contribution

In this thesis we concentrate mainly on providing environment support for requirements writing and analysis. To define exactly the problem, we introduce a characterization of requirements, as a means of communicating needs and desires about a software system, based on their linguistic properties. We also identify and characterize a number of basic activities — namely expression, understanding, modeling, validation, and metrication — that are performed on requirements in the course of a requirements process. The activities we characterize are the basic building blocks on which a process can be built: they can be found in completely informal processes and in structured processes alike.

We also define basic requirements for a "requirements engineering supporting environment", and present an extensible, component-based architecture that satisfies those requirements. As a proof-of-concept we present an implementation of our architecture, together with a number of components aimed at producing document models, system models, and process models from a set of natural language requirements, at validating those models, and at measuring their characteristics. In the course of the presentation, a number of innovative techniques and tools are introduced.

The effectiveness of the approach is discussed in two case studies, presenting applications of our environment in real cases.

## I.3 Plan of the thesis

The thesis is structured in three parts. In the first, the general field of requirements engineering is briefly presented, highlighting the facets of the discipline that are most relevant for our purposes, and presenting our conceptual framework for requirements writing and analysis. The notion of requirements engineering supporting environment is introduced, and characterized through its requirements. These are the requirements that a requirements engineer could ask for a tool supporting his or her work. The first part is completed by a survey of the literature concerning requirements engineering environments. The terms in which we have defined the problem of supporting requirements writing and analysis are novel, so most of the works cited have only a partial overlap with the work presented in this thesis.

The second part contains a detailed description of the prototype requirements engineering supporting environment we have built. We first discuss the general architecture of the system, then presents the basic functionalities of its kernel component. The architecture we describe is highly modular and extensible, and most of the functionalities are implemented in external modules. We describe in particular the module responsible for parsing of natural language requirements, that implements a novel shallow parsing technique called domain-based parsing. Approximatively other 50 modules are described, that constitute the standard library for our prototype. These covers object models, functional models, and dynamic models for a software system, as well as document models for the requirements document, and process models for the requirements activity. A number of modules are also devoted to the collection of metric data on the document, system, and process dimensions, and these are described together with some observation on their possible use.

The third and last part presents two case studies in which our framework and the techniques we developed are applied to requirements coming from industry (NASA's requirements for the International Space Station) and from an academic forum (the

Light Control System used as a common case study in a recent symposium). Both experiences show that an appropriate environment, like the one we built, contributes significantly to the construction of requirements of better quality, with less effort and with consistent and repeatable procedures.

# Part I

# Requirements Engineering Environments

# Chapter 1

# Requirements engineering


Abstract

In this chapter we characterize requirements engineering, as part of the software engineering discipline, and establish some classification and terminology of the issues that requirements engineering faces. Some general references in the area are indicated.

Requirements engineering is first presented in general terms, and then the activities that are more relevant for our work are analyzed in a little more detail.


## 1.1 Software engineering

The term “software engineering” was introduced for the first time by Farmer in 1968 [88]. Several definitions have been proposed since then, but the most referenced one is the following, by IEEE [109]:

> “Software engineering is the systematic approach to the development, operation, maintenance, and retirement of software.”

This definition encompasses a wide range of activities that cover the entire lifetime of a software system, since its inception and up to its ultimate end. Of particular interest is the first of the activities listed, the *software development process*. Many models have been proposed for this process in the literature, but the most common one is still the so-called “waterfall” model, which describes the development process as a sequence of phases producing various kinds of deliverables (text documents, diagrams or code) that are used as input to subsequent phases.

Traditionally, these documents are called *requirements*, *specification*, *design*, and *implementation* of the software product, and a substantial amount of research work has been spent on proposing different characterization of their nature. However, the distinction is not always so sharp, especially in view of advancements like executable specifications and formal language requirements. An alternative view, to which we subscribe, considers the development activity as a continuous, rather than discrete, process. Its goal is to progressively refine more abstract views of a system into more concrete ones. In this sense, the most abstract representation of a system is the (often unexpressed) customer need that the system should satisfy, while the more concrete representation is the system in itself, once specified, designed, built, and brought into operation.

In particular, requirements and specifications have traditionally been characterized by the use of informal languages and notations for the former (most often natural language augmented with drawings of some sort), and of formal languages, with a formal semantics, for the latter. The distinction, however, is blurring, and recent works advocate the use of natural language in specifications [23] while others advocate the need for formal languages in requirements documents [82].

## 1.2 Requirements Engineering

Among the various activities related to software engineering, requirement engineering covers the upper parts of the development process, and is probably the phase most affecting the result of the entire process.

### 1.2.1 Definition

Zave [114] proposes the following definition:

> "Requirements engineering is the branch of systems engineering concerned with real-world goals for, services provided by, and constraints on a large and complex software-intensive system. It is also concerned with the relationship of these factors to precise specifications of system behaviour, and to their evolution over time and across system families."

This definition identifies several key concepts that deserve some analysis. Requirements engineering is classified as a branch of systems engineering, a scope far broader than that of mere software engineering, but then it is re-specialized to the needs of a "large and complex software-intensive system". Although this specialization could be questionable when considering requirements engineering *per se* (i.e., in abstract terms), it appears adequate in our case, since requirements engineering for simple systems is often trivial and does not need special support, and systems that are not software-intensive are outside of our scope. Another interesting point is the reference to the "real-world goals" for the system under consideration. This single mention injects all the complexities, conflicts and inconsistencies of real world into requirements engineering, putting it aside from those parts of the software development process that are completely formalizable (e.g., library handling, running of test cases, or code generation). On the other hand, the reference to the "precise specification of system behaviour" clearly defines the purpose of requirements as a means to bridge the gap from the "real-world goals" to a "precise specification" of a software system.

A last interesting point in the definition from Zave is that all these factors continuously evolve in time and across system families, so that a requirements engineering process must provide enough flexibility to manage changes in the requirements at every stage of the development.

### 1.2.2 Motivation

According to Finkelstein [43], research in requirement engineering can be motivated simply by a generic concern about the quality of the resulting software system, since quality is determined exactly by the extent to which the system meets the requirements of its stakeholders. This motivation points to some of the problems still open in the area, i.e., how to identify (all and only) the stakeholders for a given system, how to determine correctly what their requirements (both implicit and explicit) are, and how to measure to what extent those requirements are satisfied by the resulting system. Moreover, the existence of multiple stakeholders calls for the adoption of processes with a number of participants, whose positions are often conflicting and unstructured. In this sense, requirements engineering borders with the field of Cooperative Work, and automated support to these activities lies firmly in the field of Computer Supported Cooperative Work (CSCW) [50, 51].

### 1.2.3 Scope

Some author (e.g., [58]) includes in the field of requirements engineering even the managerial issues of setting up an appropriate process, staffing it with adequately trained personnel, keeping records of its performance etc. However, most of these issues are not specific to requirements engineering, but rather pertain to the most general area of (business) process modeling and improvement. A more restrictive definition considers all these activities and other related ones (like contract stipulation and selection of procurement procedures) as pertaining to the *context* into which *core* requirements engineering takes place. Other preparatory activities like feasibility analysis and risk/benefits analysis can also be classified as contextual activities (although more strictly related to core requirements engineering) . Actually, there is no definite ordering between these activities and core requirements engineering: infeasibility or risks stem from infeasible or risky requirements, and still requirements engineering is considered to start *after* a project has been declared feasible and convenient.

## 1.3 The core of requirements engineering

We classify as core requirements engineering the activities of stakeholder analysis, information gathering (also called requirement elicitation), model building, validation and metrication, and documentation. These activities are described in more details in the next sections.

### 1.3.1 Stakeholder analysis

Stakeholder analysis consists in the identification or selection of the *stakeholders*, i.e., all parties interested in the development or affected by the operation of the (large, complex, software-intensive) system to design. Potentially, all the stakeholders have a right and an interest to discuss the requirements for the system, either by introducing new requirements or by validating the requirements introduced by other stakeholders. The term "parties" has been left purposefully vague, as different authors have different positions on this issue. Krogstie et al. [69], for example, consider individuals, organization or "technical actors" (i.e., tools) as participants in the requirements process, while [44] stresses the importance of the perspective or "viewpoint" on the requirements over that of the person or organization occupying it. Also, role-based analysis is widely used to abstract from the actual stakeholders when speaking of requirements processes in general, with the understood assumption that roles will be concretized by assigning responsibilities to participants in the instantiated process.

### 1.3.2 Information gathering

Once stakeholders have been identified, the difficult phase of information gathering starts. Traditionally, this activity has been seen as a conscious effort by the developer (or requirements engineer) to obtain information from the customer in order to re-formulate that information as requirements. Common expressions like "requirements extraction" or "requirements capture" refer to this narrow view of information gathering. Placing the entire responsibility for the identification of all relevant requirements onto the developer is in practice unrealistic, and has often led to process failures: for example, efforts by the developer to obtain information have been seen as an intrusive hindrance from the point of view of the customer. A more neutral view of this activity can be taken, in which stakeholders are undifferentiated, and it is assumed that all the participants have an interest and a motivation driving the identification of the "correct" set of requirements. This also means that failures are not to blame on the developer alone, but equally on all the participants [36, 91].

Information gathering covers essentially two aspects: *domain acquisition* and *problem acquisition*. The former aims at collecting all information needed to build an effective model of the environment in which the system to design will operate, as accurate as needed for the correct operations of the system. The latter aims at understanding and formulating the needs and goals of the various stakeholders, in order to define the purpose, scope and behaviour of the system to design. In both cases, information can come from a wide variety of sources, including individuals' knowledge, work practices, existing documents, accepted standards, and so on. A number of studies analyzed the effectiveness of different acquisition techniques, e.g., interviews, questionnaires, workflow analysis, user observation — see [48] for a concise review.

### 1.3.3 Modeling

After acquisition, *domain modeling* and *problem modeling* produce models of the domain and of the problem that effectively constitute the first true requirements. These modeling activities are at the core of the whole requirements engineering craft; it is through these models that the system designed to satisfy the stakeholders' goals in the given environment starts to take form. Information collected during acquisition could include conflicts that should be identified and, where appropriate, resolved during modeling. Many different methods can be used in modeling the environment and the problem, varying widely in expressive power, understandability, graphical form, scope. Among others, we can cite SADT [79], Yourdon [113], Shlaer-Mellor [103], OMT [99], and the more recent UML [14] that is rapidly becoming a *de-facto* standard as a modeling language. Specification languages like Z [105] or LOTOS [13] have also been used successfully to this end. It should be noted that many methods, e.g., UML, are parametric with respect to the language used to specify parts of the model, so various combinations are possible.

The use of Abstract State Machines [16, 18] has recently attracted much attention as a high-level specification technique. Their reduced linguistic overhead and powerful abstraction mechanism make ASMs well suited for building models of systems, and a number of case study have been published by different authors in which ASMs are used both to express the requirements for a software system and, by refinement, to give a detailed specification of the system to design.

### 1.3.4 Validation

While the gathering and modeling phases can be accurate, they cannot be perfect, so some kind of validation is needed. The goal of validation is simply to check whether all the stakeholders' requirements are satisfied by the modeled system, or in case they are not satisfied, that there is a sensible and recorded rationale for this. It is worthwhile to stress again that this covers both explicit and implicit requirements, so validation cannot be reduced to a simple check against some well-defined set of requirements.

Traditionally [12], the term *verification* has been used to designate checking that a software system conforms to its specification, and the term *validation* to designate checking that the software system solves the needs (or, sometimes, meets the expectations) of its customer. These are different properties, and correspond — in classical 1979 terms [11] — to the following questions:

- "Are we building the right product?" (validation)
- "Are we building the product right?" (verification)

In the case of software requirements, verification is infeasible, since no specification exists to check the requirements against; requirements engineering is thus concerned only with validation of requirements. The term *validation* is also used to indicate checking that a model satisfies certain internal consistency properties, under the assumption that internal consistency of a software system is always part of the customer's expectations. To avoid ambiguity, we will use the expressions "formal validation" and "user validation" to differentiate among these two senses when needed.

A number of techniques have been proposed to validate requirements documents, either methodologically (e.g., structured inspection on the requirements [38]) or as automatic tools exploiting the properties of the underlying models to ensure some desired property (e.g., by using model checking [60]). In any case, validation should be strictly tied to requirements production, to minimize the distance between the time of introduction of a "wrong" requirement and the time at which it is recognized as such and corrected. In practice, however, this is seldom possible, and requirements are validated (usually through inspection) well after the requirements document is completed. In some case, late identification of problems in the requirements can lead to significant rework on the whole document.

An important related issue is the problem of requirements *visualisation*. In order to validate the requirements, one needs to "view" what they mean. In keeping with the position expressed in Section 1.1, we consider all the embodiments of the system at different level of abstraction in a uniform way: from the customer's idea, through requirements and up to the implemented system. In this light, even non-canonical approaches like showing a prototype of the system in operation to the customer or users fall in the domain of requirements visualisation. In fact, a prototype is an effective way of dynamically visualizing the requirements that the prototype itself satisfies. In cases of more static visualisations, e.g., textual analysis or diagrammatic representations, that usually allow the inspection of only certain facets of a system, the issue of *navigation* (i.e., the ability to move in a structured way across representations) also becomes important.

It should be noted that even simple inspection has been shown to be remarkably effective [38], in particular when using automatic support tools [62].

### 1.3.5 Metrication

Tied to the validation activity, and to some extent to the subsequent verification one, is the idea that some of the characteristics of the system can be measured or estimated already from its requirements.

This field has seen little advancement beyond the use of functional metrics [65], and of readability indexes in the case of natural language requirements [29]. Also, the field of metrics for the requirements engineering *process* has not given original and satisfactory results up to our knowledge.

Still, the issue of requirements metrication is strictly tied with that of requirements quality, and given the importance of requirement quality for the overall success of a software project, it is surprising that almost no quality framework for requirements (e.g., [70, 95]) include explicit metrics.

### 1.3.6 Documentation

The requirements engineering process is a rich and complex one, and invariably it produces a wide array of heterogeneous documents. Beside traditional paper documents, both textual and graphical, more advanced media (e.g., video as proposed in [20] and [112]) are to be taken into account. While advances in database technology have made possible and efficient the storage and retrieval of such media, their use in the context of requirements engineering is not widespread. The importance of documentation is evident when considering that requirements serve as basis for verification, and are often referred to in the long term in the maintenance phase. Thus, all decisions should be traced, together with their rationale; all compromises or conflict compositions should be made explicit, and all revisions of a requirements document should be preserved. Also, links between the different kind of process documents should be established and preserved during the entire life of the project.

Also in this case, in keeping with what is said in Section 1.1, we believe that traceability should extend through the entire abstraction spectrum of the system, linking the top level requirements to the parts or structures of the implemented system realizing them.

# Chapter 2

# Environment support for requirements engineering

Abstract

In this chapter we give a characterization of requirements, in terms of their attributes and of the activities that are performed upon them in the course of a requirements engineering process. We concentrate particularly on linguistic properties of requirements (unifying in our treatment natural languages, diagrammatic languages, and formal languages) and on the modeling, validation and metrication activities.

We then introduce the concept of *requirements engineering supporting environment*, and state some of the requirements that such a system must satisfy in order to provide effective support to the activities mentioned above. The kind of support that can be obtained is exemplified through a simple scenario.

## 2.1 A conceptual framework

In this section we provide first a characterization of requirements based on their linguistic properties, and then identify a number of basic activities that are performed on requirements in a requirements writing process.

It is useful to remark that we are *not* describing a particular process here: on the contrary, since the issues related to these activities are essentially orthogonal to a specific process, we maintain a process-agnostic position in our discussion, and only define basic concepts that are applicable to different processes.

### 2.1.1 Linguistic properties of requirements

All requirements, as soon as they are expressed, assume a *linguistic form*, that is as such characterized by lexical, syntactic, semantic, and pragmatic aspects. This is true of requirements expressed in formal languages, but also of requirements expressed in natural language (NL), which are among the most common ones found in practice, and of requirements expressed in semiformal, graphical formalisms, like data flow diagrams (DFD).

*Lexical aspects* are usually tied to the applicative domain. Most often, the names used in a requirement designate relevant entities of the domain, and are substantially independent from a particular syntax. For example, the word "user" can be used equally well as a noun in a NL requirement and as a label in a data flow diagram; in both cases, the term "user" represents the same, and hopefully well-identified, real-world entity.

On the other hand, *syntax* (be it textual, graphical, or based on other media) defines the language used. In our previous example, English grammar and specific well-formedness rules constitute the syntactic part of a NL requirement and of a DFD, respectively. We also consider the notation (e.g., text, or boxes and arrows) to be part of the syntactic component of requirements.

Ideally, the *semantic content* of a requirement should be independent from a particular language, and in most cases can be communicated by different means (e.g., a DFD can be "read aloud" in natural language). Naturally, not every language can express every aspect of the semantics of a requirement (e.g., a NL requirement stating the operating temperature range for a device cannot be expressed as a DFD).

A substantial amount of information in requirements is also carried via their *pragmatic aspects*, e.g., by the particular layout of a diagram or by the choice of particular wording in a NL requirement. Most often, these aspects reflect the writer's desire to express more information than the language in use allows, in a sort of out-of-band signaling style.

### 2.1.2 Basic activities on requirements

Of a (possibly formally defined) requirements engineering process, we are interested in observing only the repeated exchange of requirement proposals among the participants, leading in time to (1) a set of requirements acceptable to all the participants, and (2) a set of residual conflicts between stakeholders, together with their rationales. This exchange can be decomposed into two basic activities:

- **Expression.** Requirements are at first expressed by a stakeholder. This activity associates a syntactic and lexical form to the requirements semantics (i.e., what the stakeholder wants to express). The syntactic form provides a frame for the semantic content, while the lexical form links the requirement to some set of agreed-upon concepts. After expression, requirements have a physical form, and are thus amenable to transmission and processing. In particular, they can be communicated to other participants to the requirements writing process. Expression also injects some pragmatic information to go with the syntactic and lexical form of the requirement.

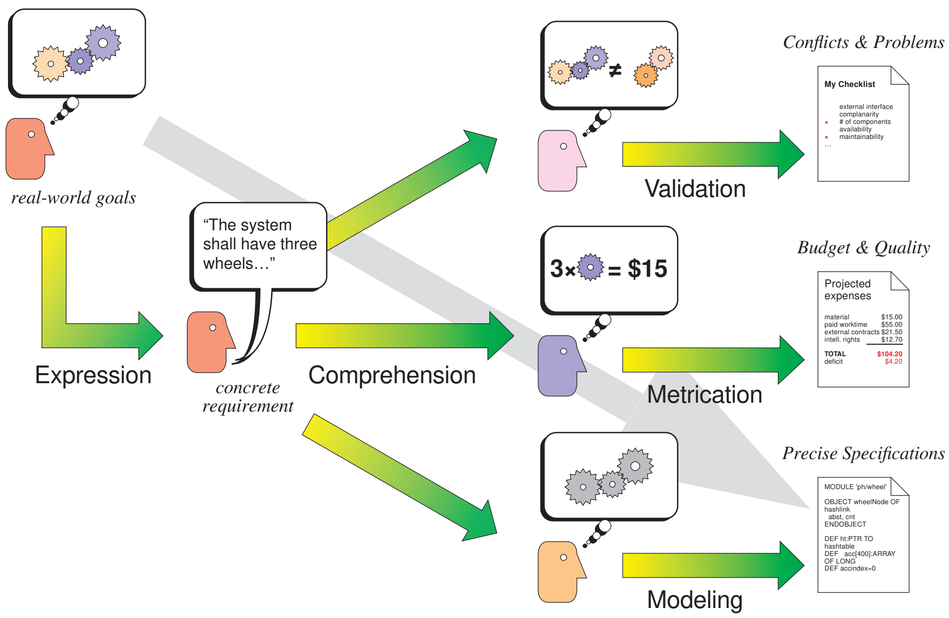



Figure 2.1: Main activities performed on a proposed requirement.

- **Comprehension.** Requirements are then understood by other participants. In our framework, understanding refers to the operation of extracting the semantic content of a requirements from its syntactic and lexical form, by using some base knowledge of the participant.[1] Part of the semantic content can also be extracted from pragmatic information.

It can be easily understood that we have correct and complete communication only when the semantic content of a requirement, originally present "in the head" of a participant, can be reconstructed identical "in the head" of a different participant, as shown in Figure 2.1.

Incorrect communication can be due to the inability of the stakeholder expressing the requirement to encode all the semantic content into the syntactic, lexical and pragmatic form, or to the inability of the stakeholder understanding the requirement to extract all the semantic content from its syntactic, lexical and pragmatic form.

Of these, the lexical form is the one that must necessarily be shared. Without a common lexicon serving as a shared ontology, stakeholders could not communicate at all. Notice that even a very small initial common lexicon can be used to build upon it more complex ontologies. For example, a good start at learning the vocabulary of an unknown language is to recognize that pointing at something with the index finger and yelling something is a sign meaning "that sound is the name of this thing". Even sharing that single sign (a deictic) can be enough to build a richer common lexicon. Simple means to ensure that all the stakeholders share a common lexicon are in widespread use. For example, the vast majority of requirements documents in industrial usage have a "Dictionary of Terms" section listing and defining each sign in the domain (in terms of other already-known signs).

Sharing of syntactic form can be partial. For example, a requirements document could contain natural language descriptions of the behaviour of a certain component side-by-side with a formal Petri net specification of the same behaviour. The two representations taken together constitute a single richer language, but knowledge of only part of its combined syntax is sufficient to extract at least a part of the semantic content. Well known means exist to ensure sharing of syntactic knowledge, voluminous manuals and textbooks being the most common example. In our previous example, to ensure knowledge of Petri nets syntax and semantics a description of the formalism could be added to or referred in the requirement document. Notice that such a description would at first introduce simple concepts by using an already shared language (e.g., natural language), and gradually build on that to ensure understanding of the new formalism.

Finally, sharing of pragmatic forms is made difficult by the informal nature of pragmatic content. By definition, pragmatic aspects are never codified (since as soon as they are codified, they become part of the syntax of a language), so sharing of pragmatic forms can only be obtained by informal means like experience and aesthetic judgement. This is the reason for recommending explicitness whenever

[1]Note that we are *not* asking here that this knowledge is shared by all participants.

possible: in our framework, making something explicit means encoding it in syntactic and lexical terms.

### 2.1.3 Other activities

Communication, however, is only a step of the whole process; the success of the mere communication of requirements *per se* is not sufficient to ensure the success of the requirements process. In fact, comprehension leads to three other activities (see again Figure 2.1):

- **Modeling.** Different stakeholders tend to reason about the system to design in terms of different abstractions, varying in depth, detail and focus, and according to their goals and interests. Different abstractions correspond to different models, so the ability to build models is essential for an effective participation in the requirements process.

- **Validation.** When considering a candidate requirement proposed by another participant, each stakeholder implicitly performs several kind of validations, e.g., internal consistency of the proposal, or compatibility with the stakeholder's goals. The outcome of this validation step determines the acceptability of a proposed requirement, or exposes conflicts among the different viewpoints.

- **Metrication.** A number of technical and management decisions depend on the ready availability of different measures on the requirements, on the system described by the requirements, and on the requirements process itself. A stakeholder must be in a position to measure or estimate relevant attributes in order to take informed decisions during the process.

It is important to remark that all these activities are *not* sequential steps of a well-defined process. In the vast majority of cases, they occur as mental processes of the participants in a completely uncoordinated fashion, the only synchronization points being the communication of proposed requirements, amendments, rejections, counter-proposals, etc.

It is also worthwhile to notice that there is a rich connective tissue among the three activities described above. In fact, validation is often performed by checking desired properties on some model, while metrication can measure the attributes of a model in order to measure or estimate attributes of the requirements. On the other hand, the outcome of a validation activity can constitute a model on its own (e.g., a model of the problematic areas in a requirements document), as can the outcome of a metrication step (e.g., an instance of a quality model). It is also possible to collect metrics on the results of a validation activity (e.g., how many residual errors are in a document), or validate a set of metric data against some acceptability profile.

Rich connections, however, do not imply interchangeability. While a certain set of data can be seen as a measure from a stakeholder and as a model from a different on, the activities that are carried by the stakeholders on those data are still distinct and well characterized.

## 2.2 Requirements Engineering Supporting Environments

### 2.2.1 Problem statement

The particular problem we focus on in this thesis, with reference to the framework we presented above, is to provide environment support for the expression, comprehension, modeling, validation and metrication activities.

Environment support means that all these activities must be supported in a uniform way, and that to a user working in a supporting environment, the application of tools, methods and techniques providing support must constitute a natural extension of his or her mental processes.

We are also concerned with the issue of presenting different views on the requirements to different stakeholders, according to their knowledge, background, goals and to their role in the requirements engineering process. While this is essential when different stakeholders do not understand each other's "language", it is also of great benefit in simple cases where interaction is very limited. In fact, working with more focused views greatly simplifies the correct definition of particular aspects of the system to design.

We are *not* concerned with establishing a process on top of our environment. We believe that defining, modeling, enacting and controlling a process is an issue that is orthogonal, and of a much higher level, to the ones we are interested in. We study here which support can be offered to some basic activity (e.g., expressing and understanding requirements); these basic activities can be organized in a rigorous process, but can also be part of an anarchic, uncoordinated design activity.

### 2.2.2 Requirements for RE supporting environments

In order to support the expression and understanding activities, a support system must be able to accept requirements in a certain syntactic, lexical and pragmatic form (expression), and to offer the same semantic content packaged in a potentially different syntactic and pragmatic form[2] (understanding). This implies that the supporting system must have the ability of separating all the linguistic components of a requirement and of synthesizing new forms carrying the same semantic content, either completely (translation) or partially (translation with focus on certain aspects). This problem is sometimes referred to as the problem of offering multiple views on the requirements.

[2]Notice that we assume that the lexical content remain unchanged, see the discussion in Section 2.1.2.

Moreover, a support system must provide means to build abstractions (models) on various aspects of the requirements, and to validate and measure such models.

We state thus the following requirements for a RE supporting system:

1. **The support system must allow the expression of models of the system to design and of its environment, in a way that is independent from a particular syntactic form**.

   This requirement states that users of the system must be able to express their requirements in the syntactic form that is most adequate to the requirements and to their knowledge and capabilities. In other words, a supporting system should not force a non-technical customer to learn formal languages, nor should it force a requirement professional to express requirements in inadequate languages when more expressive languages are available.

2. **The support system must be able to store and retrieve expressed models**.

   This is a general requirement to ensure the persistency of models, both to allow non-synchronous interaction among stakeholders (short term persistency), and to support version and configuration management, documentation, and requirement tracing (long term persistency).

3. **The support system must be able to synthesize models of the system to design and of its environment, using stored information, in a way that is independent from a particular syntactic form**.

   This is the dual of requirement 1. Together with its dual, this requirement specifies the full or partial translation functionality. The reference to *models* of the system to design implies the capability of building abstractions on the models expressed.

4. **The support system must maintain the lexical content of the models, and should preserve pragmatic aspects as far as possible**.

   Maintaining lexical content means that the names of domain entities should not change among models and representations. Moreover, if a renaming is asked for, it should affect all the models and representations, so that at each instant, only one well-defined name exists to designate domain entities.

   Requesting the preservation of pragmatic aspects implies that the supporting system must try to convey the same information carried by pragmatic information in different representations. For example, if a data item name is underlined in a certain passage of a natural language requirements document, the corresponding data store in a data flows model could be highlighted by drawing it in a different color. This kind of mapping is particularly difficult, so preservation of pragmatic aspects is only an optional requirement (as indicated by the optative mood).

5. **The support system must be able to perform a formal validation of stored models, and must support the verification of properties of such models**.

   This requirement asks for both the formal validation of models, i.e., for the check that a predefined set of properties that characterize valid models hold on a certain instance of a model, and for the verification of arbitrary properties provided by the users of the support system. Notice that queries of different kinds on the requirements base can be considered as checks that the property "the requirements base contains data that satisfies the query" holds.

6. **The support system must be able to extract metric data about the system to design and about the requirement process itself**.

   We have already discussed the need for metrication in a requirements engineering process. This requirement states that a supporting system should be able to compute such data from models expressed, stored, and synthesized (according to requirements 1–3).

7. **The support system must provide suitable user-level representations (views) for models, validation reports and metric data**.

   Models are abstractions of concrete entities. This requirement states that the supporting system must be capable of associating a concrete representation to abstract models and to the outcome of the validation (requirement 5) and metrication (requirement 6) activities. Moreover, these representations must be suitable for presentation to the users of the supporting system, taking into account the potential presence of non technically educated stakeholders.

   This requirement does not ask that *all* representations should be suitable for *all* users, but that for *each* user, *some* kind of suitable representation must be provided by the system. Indeed, elementary representations (e.g., long lists of textual material) could be considered not suitable for highly skilled users of the supporting system.

8. **The support system must allow the easy integration of arbitrary models and representations**.

   No arbitrary set of languages, models, representations, validation and metrication techniques can ever be considered complete. For this reason, a supporting system must be easily extensible (possibly by the stakeholders themselves) to adapt it to the needs of a certain project, or to the skills of a certain set of stakeholders.

   Easy addition (providing new capabilities to the system) and integration (ensuring that the new capabilities benefit from already-existing capabilities, and vice versa) of capabilities also allow an organization to incorporate its capital

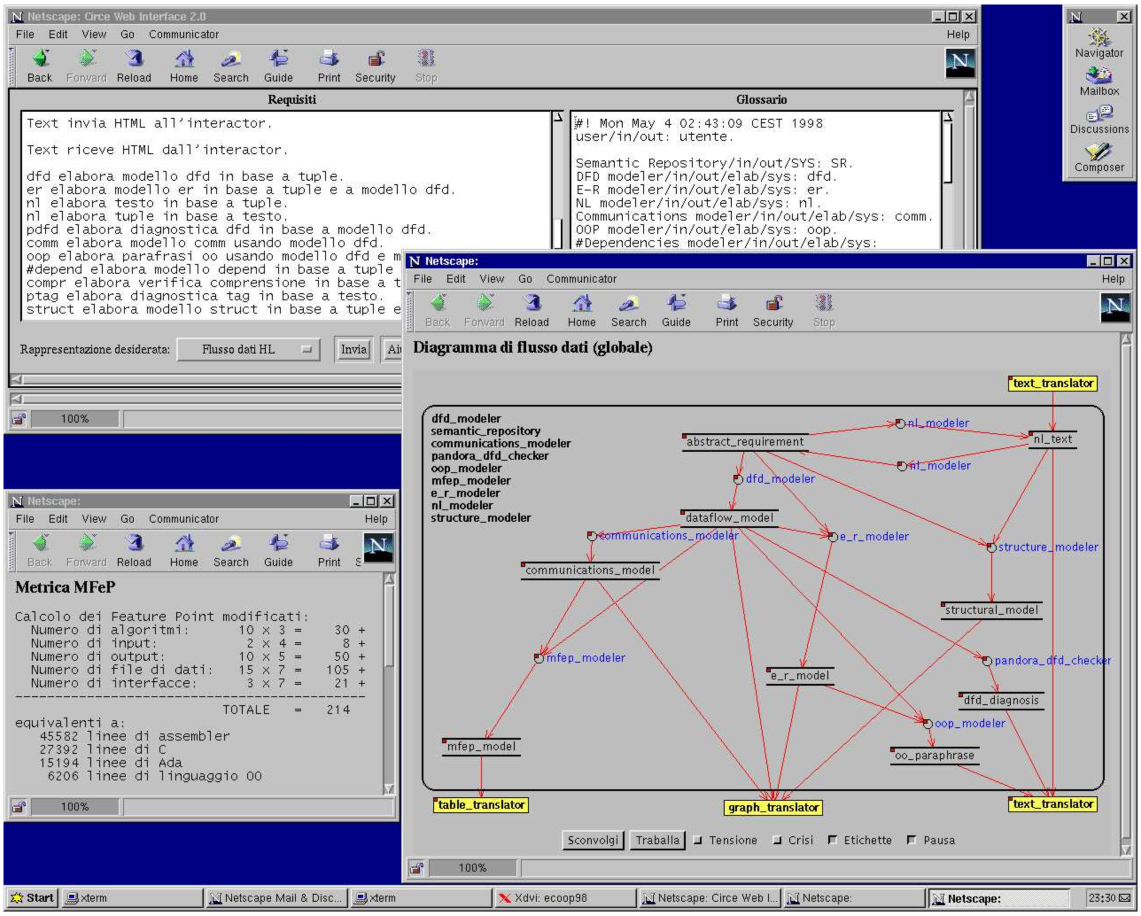



Figure 2.2: A session with CIRCE. Three views on the same requirements are shown; from the top in clockwise order: NL requirements (in Italian), data flow diagram, functional metrics.

of practices and expertise into the particular instance of the support system in use at the site.

While these requirements are certainly not exhaustive, we believe that a system satisfying them will be able to perform the basic functions needed to offer the type of support we are looking for.

On the other hand, systems *not* satisfying one or more of these requirements will present significant shortcomings when used in the context we have outlined above. See Chapter 3 for examples of such systems.

### 2.2.3 A simple scenario

Figure 2.2 shows a work session with our requirement engineering environment CIRCE (described in details in Part II). On the top, an editable text field allows the user to express his or her model of the system being specified by writing NL requirements (per our requirement 1). The environment can parse the syntactic form (in our case, natural language) and store the model obtained (per our requirement 2). It can then synthesize new models from the original description, e.g. a data flow model of the system.

Synthesized models can be presented via different syntactic forms (requirement 3), e.g. textual, tabular, or graphical — as shown in the DFD window in Figure 2.2 —, and can themselves be editable. The lexical component is preserved in all representations. For example, the nodes of the DFD diagram are labeled with the same nouns as used in the NL description (requirement 4).

Models can be validated against a set of desired properties, and their attributes can be measured (requirements 5 and 6). In any case, a suitable user-level representation is provided (requirement 7). In our example, the functional score for the system described by the requirements is computed, and presented to the user in a palatable format (bottom left corner window in Figure 2.2).

In a typical case, some non-technical stakeholder (e.g., a customer) could write a first NL requirement document, thus providing an initial sketch of the system to design. Another stakeholder, more technically educated (e.g., a developer), could look at the corresponding data flow diagram and spot some missing flow or computation. The developer could copy and paste this view into an e-mail, discussing the issue with a colleague, and then simply add the missing element by editing the diagram: the change would be propagated to the first stakeholder's view of the system, by adding the corresponding synthesized NL statement to his or her document. The customer could then easily confirm whether the developer's understanding leading to the modification was correct or not with respect to his or her own point of view.

Meanwhile, a third stakeholder (e.g., a project manager) could be interested only in the functional score of the specification, to ensure that the projected cost for the system does not exceed allocated budget. Requirements evolution could continue in a similar way, exploiting the combined work and competencies of all the participants, until a stable set of requirements is obtained, and conflicts have been solved by mutual agreement or by political decisions.

Incidentally, the system being designed in the session captured in Figure 2.2 is an early version of CIRCE itself.

# Chapter 3

# Previous work

Abstract

In this chapter we present a concise survey of the literature pertaining to requirements engineering environments. By their very nature, environments encompass many different aspects of a problem. In our survey, references have been grouped according to their principal contribution. Some work not specifically discussing environments for requirements engineering, but presenting proposals significant for the field, is also included.

## 3.1 User participation

The need and effects of direct user participation in requirement analysis have been considered since the early '90s, e.g., Brown [19], Drake et al. [31] and Cucchiarelli et al. [25]. In a more recent work, El Emam et al. [35] present a statistical model, derived from empirical data, in which the participation of the user is a factor reducing uncertainty on the outcome of the development process. Keil & Carmen [67] have scrutinized 31 industrial projects at 17 companies, identifying 15 different customer–developer types of links and concluding that projects with more and more direct links have been more successful than those with weaker links.

The view that requirements engineers should *not* become expert in the domain and problem at hand is taken by Berry [10], among others; on the other hand, Nuseibeh [90] warns against excessive reliance on the user's view of the problem. The need for validation from the user is recognized since a long time; interestingly, in an empirical assessment of the effectiveness of different design methodologies upon user's comprehension, Nosek and Schwartz [89] found no evidence at all that formal representations had any advantage over narrative text.

## 3.2 Informality vs. formality

The debate about whether formality is needed in requirements and specifications is still open, although most researchers and practitioners alike concede that formality offers the most important advantages. Lawrence in [72] and Jackson in [59] champion the sides of informality and formality, respectively, in a pair of related papers. Lawrence's position is essentially that informal requirements are easier to understand, and thus more effective in practice, than formal ones. Jackson's position, instead, is that formal languages and methods help in exposing subtle problems, in abstracting the problem, and in analyzing it through mechanical aids. The fact that the debate is still open since at least twenty years (see for example [8] and [81] from the late '70s) is a hint that both positions have valid arguments on their side.

The need to bridge this gap was expressed by several authors, e.g., Fraser et al. [47] and Bubenko et al. [21]. The common baseline is that informal and formal requirements offer complementary advantages (e.g., understandability vs. precision), and thus the best of both worlds can be obtained by using some method to switch forth and back between the two. In particular, in [47] a methodology is proposed to transform informal requirements (à la Structured Analysis) into formal ones (VDM). Similar approaches are described in [71] and [52].

## 3.3 Languages

In an attempt to present formally modeled requirements in a way that is understandable to the user, so-called "explaining systems", i.e., systems capable of producing a natural language version of the model, have been designed [28, 106, 107]. However, no evidence is offered of the usefulness of such systems in practical use. In fact, while explaining systems can help in obtaining a user-level validation of the requirements, the most difficult issues, i.e., information gathering and modelling, are still left to the experience of the requirements engineer.

The opposite approach is taken by a number of works, whose aim is to automatically derive a formal model of the system to design, starting from a set of informal requirements. Typically, "informal requirements" is interpreted as "natural language requirements"; up to our knowledge, no other kind of informal requirements (e.g., drawings) has been analyzed. Most of the work in this field is targeted at information systems and business applications: for example, Rolland and Proix [22] present a system capable of extracting a conceptual model through techniques allegedly similar to the ones used by a person reading natural language requirements. Interestingly, the system presented can use natural language generation techniques to create a paraphrase of the original requirements that can be then validated by the user.

Macias and Pulman also used linguistic techniques to pursue similar objectives. The system described in [76] uses a well-defined subset of English to forbid the expression of potentially ambiguous sentences, and can translate natural language sentences in this reduced language into formal language and vice versa (by using a general-purpose, domain-independent NLP engine). In addition to paraphrasing, the system described supports natural language queries to the formal model so built, whose answers are themselves expressed in natural language (such queries have since long been studied in the database field, e.g., [66]). The same authors also present in [77] a syntax-driven interface that prevents the creation of "incorrect" requirements, according to their definition of correctness.

## 3.4 Analysis and validation

The idea that requirements could be analyzed in an automatic fashion in order to identify and possibly correct inconsistencies, redundancies, incompleteness, and other defects has attracted much attention.

Among the older works, Boehm [12] indicated that errors in requirements are the most costly in the whole software development process, citing a two orders of magnitude ratio between the cost of correcting errors in requirements and that of correcting errors in actual code. Since then, a number of papers have appeared dealing with the problem of identifying errors in requirements.

Heitmeyer et al. [54, 56] present a method based on the Four-Variables Model by Parnas and Madey [94] to discover inconsistencies in a requirement specification. The authors also discuss the design of a prototype tool and give conditions under which the consistency checks can be complete. However, the method requires that requirements are expressed in a formalized notation, based on finite state machines and event/condition tables. The requirement process envisioned by the authors, starting with the phrase "A formal notation, such as the SCR notation, is used to specify the requirements", still presents all the difficulties tied to information gathering and modeling. Only *after* these activities have been completed can a requirements engineer write down the model he or she has built using some formal notation. This is especially true of notations designed to be easily analyzable, and not targeted at being a practical language for building and experimenting with requirements.

Jackson and Damon [60] used a similar approach in their tool "Nitpick". In their case, requirements are expressed in a subset of Z, and the tool can check the specification in order to find a counterexample with respect to claims stated in the same language. By using sophisticated model checking techniques, Nitpick can validate a model in only a few seconds. Obviously, other model checkers could be used in the same way (e.g., SPIN [57]), but as the language used becomes of lower level (Promela, in the case of SPIN), the tools become less suited to the requirement phase.

Duffy et al. [32] tried to integrate formal and informal representations of the requirements by requesting that *both* must be stated side-by-side; analysis is then performed by using only the formal representation. However, equivalence between the two representations of the same requirement is assumed, and no guarantee is given of their actual correspondence. So, effectively, their proposal amounts to annotating a formal requirement document with abundant natural language comments, and offers no real advantage over the ones presented above.

A different approach is taken by the older "Requirements Apprentice", by Reubenstein and Waters [97, 98]. The Requirements Apprentice is a support system whose goal is to assist humans in validating requirements. Interestingly, in contrast with most other solutions, the Requirements Apprentice does not stress the fact that requirements must be expressed in a formal language to start with, but rather tries to facilitate the transition from informal to formal requirements by using a natural language-like syntax. Also, the system uses some knowledge about the applicative domains (in the form of so-called clichés) to validate the requirements. Internally, the Requirement Apprentice uses a standard frame-based knowledge repository both for clichés and for requirements. Operating on this representation, a hybrid reasoning system called Cake [40] can detect contradiction and incompleteness, disambiguate ambiguous sentences, and provide natural language output, thus producing actual requirement documents.

In the presentation by Reubenstein and Water, however, there is a problem hidden under the label "natural language-like" syntax. Actually, commands and requirements are to be entered as LISP expressions, and only a careful choice of names makes these expressions appear like (parentheses-ridden) natural language. It is not by accident, thus, that the envisioned model of usage places particular importance on the role of the requirements engineer as interface between the customer and the supporting system: the Requirement Apprentice is meant only as a support tool for the analyst.

Also, the degree of specialization required to clichés is quite high: in the example provided, concerning a library, the existence of a cliché for "lending-repository", "borrowable-object", etc. is assumed, and the requirements seem to just bind the name "library" and "book" to these clichés.

A more recent work by Maiden [78] follows a similar approach, but with a more relaxed view of clichés (here called "problem abstractions"). These problem abstractions, organized hierarchically, have a direct connection to the natural categories that, in cognitive sciences, are assumed to categorize the memory and knowledge of a person. The 13 abstraction trees (accounting for two hundred leaf nodes) proposed by Maiden are then mapped into linguistic constructs and used as a basis for the AIR Toolkit. Among other tools, the Requirements Refiner allows for a kind of structured editing that drives requirement writing by considering each requirement as an instance of one of the known abstractions. For such requirements, a number of interesting properties hold by construction. A second tool, called the Requirements Critic, performs validation of the requirements via an interactive dialogue with the user.

## 3.5 Presentation and navigation

Many of the tools proposed for requirement analysis and validation are somewhat involved with the issue of requirement presentation. However, while some of them only offer just sketched interfaces (as we pointed out in the case of the Requirement Apprentice), other authors have devoted much more attention to the subject.

Johnson et al., in [63], discuss the approach taken to requirement presentation in their tool ARIES. Requirements and domain description statements are entered into the system by using a declarative language, and stored in an internal knowledge base. ARIES can then present this knowledge to the user either in textual form (by using the GIST explainer [106, 107]), or in graphical, diagrammatic form. Their work is original in that the diagrammatic form can be directly manipulated by the user by applying a number of *stereotypical operations* (ARIES provides over 180 operations) that propagate their semantics to the underlying knowledge stored in the knowledge base. For example, a given gesture in a finite state automata representation could indicate the define-and-check-enabling-state prototypical operation, that in turn records the corresponding relation between states in the knowledge base.

A last interesting point in the work by Johnson et al. is that the representations and operations provided by the system are represented in the very same knowledge base, so that the system can be modified and extended by operating on its self-representation through ARIES itself.

Most of the research on requirements navigation has concentrated in providing hypertext capabilities to requirements document. Almost invariably, only natural-language requirements are considered; hypermedia extensions are sometimes indicated as future works.

Given that hypertext technology is quite independent from requirement engineering issues, the main contribution of these works has been to define a number of relations that can subsist between requirements and requirement elements. For example, Pohl and Haumer [96] individuate, from a survey of the literature, seven types of nodes (e.g., atomic requirements, set or sequences of requirements) and five types of links (condition, content, documents, evolutionary and abstraction), divided into 18 finer categories (example_for, comment, formalizes, refines, etc.). An editor/browser called PHEd, based on their model, is offered as a means of authoring and browsing requirements.

A similar approach was taken by Cybulski and Reed [26], that extended their interest also to other phases of the software development process.

Recently, Taivalsaari [108] pointed out the usefulness of multidimensional browsing, i.e., hypermedia browsing in which each object is linked to others along several different dimensions that the user can freely change during navigation. Although the tool presented by the author is aimed at the design phase and is suited to the visualisation of design- or implementation-level entities (e.g., code modules), the principle could be applied to requirement- and specification-level objects as well.

## 3.6 Cooperation and conflicts

The main body of work on cooperation and conflicts in requirement engineering has been devoted to the study of so-called *viewpoints*.

According to Finkelstein et al., a viewpoint is the combination of a "participant" to the software development process with a "view" or "perspective" that the participant maintains on the problem [44]. The first reference to the concept is probably that in the 1979 work by Mullery [84] on the CORE method for requirement expression, further refined in [85]. More recently, Finkelstein, Nuseibeh, and colleagues have analyzed the various issues related to viewpoints in a number of papers: [93] constitutes a general introduction, [33, 44, 104] discuss the nature of overlaps and inconsistencies among specifications, and methods to identify them, and [92] presents a tool called "The Viewer" that supports viewpoints-based requirement engineering.

In these works a viewpoint is characterized as a 5-tuple composed of:

- a Representation Style, i.e., a preferred notation;
- a Work Plan, i.e., the development strategy adopted;
- a Domain, detailing the part of the problem domain the viewpoint is interested into;
- a Specification, relative to the viewpoint's point of view and expressed in the chosen Representation Style;
- a Work Record, i.e., an history of the development activities carried on by the viewpoint on the Specification, according to the Work Plan.

In particular, the approach proposed in [44] to identify and handle inconsistency is based on the uniform representation of the views of the different stakeholders into *some* logic (potentially different for every pair of views), and on the use of meta-rules specifying which actions should be taken if an inconsistency of a certain kind is discovered between the views. The paper also proposes an example based on a temporal logic representation of different views on the requirements for a library. However, it is not clear whether a common logic can be found for every pair of viewpoints, whether the process of finding a common representation is automatizable, or how much effort is required to find ad-hoc logics.

On the other hand, Zave and Jackson [115] propose the use of a *single* logic for all the representations, and consider composition of partial specifications as simple conjunction of assertions in the logic. A set of partial specifications is thus declared consistent only if their composition is satisfiable. In the case of this idea, the dual objection holds, i.e., it is not clear if a single logic can be expressive enough to account for every possible viewpoint. Also, Zave and Jackson do not consider the issue of handling the inconsistency after it has been detected — this is left to some unspecified extra-process mechanism. The same idea of a single underlying representation (here called *canonical representation*) has been proposed by Meyers and Reiss in [83], with a similar outcome.

Both approaches presented above take the view that inconsistency should be tolerated during the development process, as long as it can be detected when needed (i.e., before committing the final requirements document). Older works with a similar position are [102] and [7], with some formal foundations provided by [27]. It is worthwhile to notice the difference between this view and the opposite one, stating that consistency should be *enforced* during the requirement process. Most of the tools and methods we discussed in Section 3.4 take the latter view; this can probably be ascribed to the intended use of such tools by a single requirement engineer, so inconsistency can arise only from "errors" in the specification. On the other hand, viewpoints-based tools are naturally targeted at cooperative requirements engineering, so inconsistency can arise as result from different views of the various stakeholders about the system to design.

Naturally, a huge amount of research on cooperation has been produced by people in the CSCW and business process re-engineering fields. Although many of the proposals and results from these researches could be applied effortlessly to cooperative requirements engineering, they are not specific to the field, and we will not discuss them here. An account on the state of the field can be found in [9] or in [110].

## 3.7 Metrics and quality models

As anticipated in Section 1.3.5, not much attention has been devoted to the issue of requirement metrication; not at least in the rigorous sense, as pointed out in [42]. When speaking of quality of requirement documents, most published works can give good advice, usually taken from direct experience or based on anecdotal evidence. Although this kind of advice can be useful in managing real processes, it is of scarce scientific interest — unless used as some kind of "experimental data".

Several quality frameworks or models have been proposed, of varying degrees of formality. Among others, the two models of Pohl [95] and Lindland et al. [75] deserve some discussion. Pohl's framework consists of three dimensions:

- the *specification dimension* (from opaque to complete requirements),
- the *representation dimension* (from informal to formal languages), and
- the *agreement dimension* (from personal to totally shared views).

Each dimension is analyzed from an input-output perspective, and the goals of different typical stakeholders are positioned in this three-dimensional space. According to this framework, requirement quality can be seen as the distance, in the space described above, between an actual requirements document and a stakeholder's goal.

The model proposed by Lindland et al. distinguishes *syntactic*, *semantic*, and *pragmatic* qualities. Syntactic quality concerns the correctness of a requirement document with respect to the language it is supposed to be written in; semantic quality concerns the validity and completeness of the requirements with respect to the "perfect", ideal description of the domain and the problem; pragmatic quality concerns the degree of agreement that the stakeholders reach on the requirements (including misconceptions as well as legitimate different goals). Moreover, three mutual *appropriateness* relations are defined: between language and domain (the language should be expressive enough to express properties of the domain), between language and participants (participants should be proficient in the chosen language), and between participants and domain (participants should be familiar with the domain at hand). A number of traditionally recognized qualities are then classified according to this framework.

In a more recent joint work [70], these two models are shown not to be in contradiction, and a unifying framework is proposed, essentially by refining Lindland's model to take into account a more detailed view of social and cognitive processes (e.g., the introduction of an explicit representation of the participants' knowledge).

# Part II

# Circe: a RE-Supporting Environment

# An introduction to CIRCE

In this part of the thesis we present CIRCE, a prototype requirements engineering supporting environment we designed and implemented. CIRCE is first described in terms of its general component-based architecture, then, more formally, by specifying the different classes of components and describing the core operations performed by its kernel.

We then describe a number of modules that constitute the basic library of CIRCE components, presenting the models they build from user requirements, and discussing how they can be used to improve the user's analysis capabilities. One of the most important modules, our natural language parser CICO, is described in more detail, due to its role as the main input method for requirements in CIRCE. While the architecture supports the expression of models and of requirements through different representations, we concentrate on natural language requirements since these are the most common requirements found in practice.

CIRCE has been used with promising results in a number of cases, both academic and in cooperation with industrial partners, in projects of small to medium dimension (up to 250 requirements). In most of these cases, users of the system deemed tool support to be a useful addition to their RE processes and showed satisfaction with the performance of the tool.

# Chapter 4

# System architecture

Abstract

In this chapter we present the general component-based architecture for a requirements engineering environment. Five kinds of components constitute the architecture: repositories, modelers, projectors, translators, and interaction modules. We show how the proposed architecture satisfies the requirements we stated in Chapter 2, and give a formal specification of each kind of component of our architecture. A discussion on the dynamic nature of the architecture and on the role of reflexive components follows the presentation.

We also describe CIRCE's kernel, the core of the system responsible for the basic functions of our environment. Kernel's function include scheduling and controlling the execution of other components (modelers, projectors, translators), providing authentication and authorization for the users, and low-level maintenance of the repositories.

## 4.1 Overview

In Chapter 2 we introduced the separation between the semantic content of a requirements document and the lexical, syntactic, and pragmatic content. In CIRCE, this separation is reflected at the architectural level by different kinds of components.

As can be seen in Figure 4.1, CIRCE's architecture is based on five kind of components. At the basis we have a central knowledge repository, that we call *semantic repository*, storing the semantic information input as requirements by the users of the environment. The semantic repository stores very basic semantic information fragments, of a fine granularity; we call such fragments *atomic requirements*. Examples of atomic requirements are a single arc in a data flow diagram, or the existence of a single attribute for an object in an entity-relationships diagram. It is worthwhile to stress here that the semantic repository does *not* necessarily contain a correct, consistent set of requirements. On the contrary, it stores every single fragment of

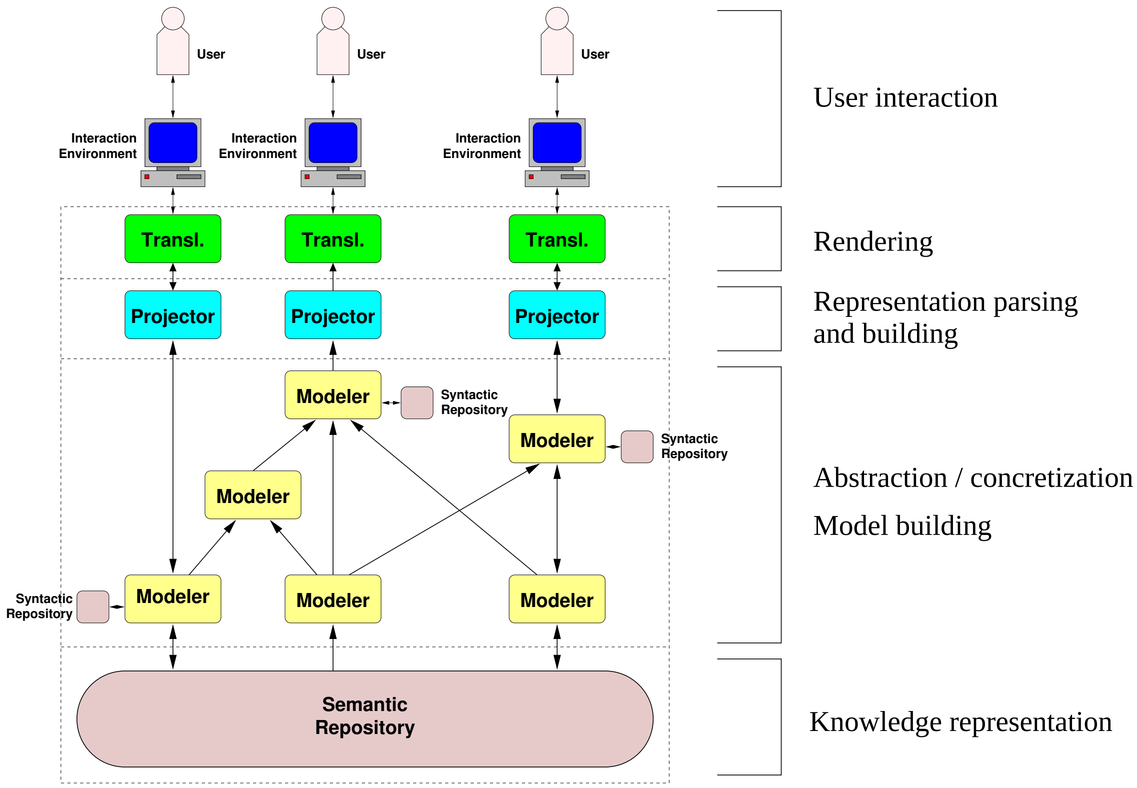



Figure 4.1: Reference architecture for a RE supporting system.

information input by every user of the system, leaving correctness issues to the relevant validation modules.

Model building, validation and metrication are performed by specialized modules called (with some bias) *modelers*, arranged in a network structure. These components perform two basic functions in our architecture. Their main purpose is the synthesis of a model starting from atomic requirements and fragments of lower-level models. This can be seen as the processing of information flowing upward in Figure 4.1. In the case of *editable* models, a modeler can also decompose models input by the user into atomic requirements, to be stored in the semantic repository, so that knowledge acquired via a model can be profitably used in building different models. In this case, too, a modeler can use lower-level modelers to gradually decompose a complex model into atomic requirements. This corresponds to the processing of information flowing downward in Figure 4.1.

In abstract terms, modelers capture part of the knowledge of the requirements engineers who wrote them about how different models of a system relate each other, how inconsistencies could be spotted, or what metrics aspects are significant, and for what purpose.

A number of *syntactic repositories* are used in connection to modelers to store information on the lexical, syntactic and pragmatic aspects of a model. This information can prove useful in re-synthesizing a model that was originally input in the system via the same representation.

Other modules called *projectors* constitute the third class of components in our architecture. Projectors are in charge of building an *effective representation* of a model (that, up to this point, exists only in abstract form) and of parsing representations input by the user. Users can only introduce or examine models via one of their representations, as produced by a projector.

To maintain a certain independence from the visualisation layer, projectors do not directly produce rendering instructions; rather, they describe views in an intermediate language that is interpreted by a *translator* module. Every translator defines a sub-language for a specific class of representations (e.g.: graphs, tables, histograms, etc.), and translates the description of a view given in the sub-language into rendering instructions appropriate for the operating environment.

Actual user interaction, that can be as simple as the reading of some text, or as complex as the editing of a complex graphical diagram, takes place in an *interaction environment*, potentially different from user to user. The interaction environment handles graphical rendering and user interaction, and provides a graphical interface appropriate for the operating environment in which Circe is installed.

A *configuration* of our architecture is obtained by populating the modelers, projectors, and translators libraries of a system with specific components, and by providing a suitable interaction environment. A full *instance* of the system is obtained from a configuration by specifying the contents of the syntactic and semantic repositories. This association is usually established dynamically for each user and project, as described in Section 4.5.3.

The operations of the system can be described in general terms as follows (see Section 4.5 for a more rigorous definition). When any of the users of the system wants to express a requirement, he or she associates mentally a syntactical and pragmatic form to the semantic content that constitutes the mental model of the requirement. Through the interaction environment, that provides appropriate editing functions (e.g., a text editor, or a graph drawing program), the requirement — in its physical form, i.e., lexicon, syntax, and pragmatics — is passed down to a translator. The translator converts representation-specific information (e.g., a sequence of strings, or a set of coordinates for a drawing) into an internal language describing the logical structure of the information (e.g., a markup language).

Based on the kind of syntactical form given to the requirement by the user, a projector is then called upon to extract the semantic content from the syntactical form. The result of this translation step (an abstract model) is passed down through a network of modelers that decompose the model into elementary fragments (atomic requirements), that are finally stored in the semantic repository. In the process, lexical, syntactic and pragmatic content may be stored in syntactic repositories.

Once the original requirement has been translated, fragmented in atomic requirements[1] and stored in the semantic repository, it can be exploited by other users. Whenever a user asks the system for a particular view on the requirements, for validation checks, or for metric data, a request is sent through the interaction environment to synthesize a particular model. All the modelers that can contribute to the requested model cooperate in extracting relevant atomic requirements from the semantic repository and in synthesizing progressively more refined intermediate models (during this process, modelers can also access their semantic repositories to obtain further information). In the end, the model requested is collected from the modelers and passed to a projector that produces a concrete representation for the model. The representation, encoded in an internal language, is then finally converted by a translator into rendering instructions for the interaction environment, that can show the requested view to the user.

## 4.2 Satisfaction by construction

In this section we show how the proposed architecture satisfies the requirements we identified in Section 2.2.2.

1. **expressing models.** Thanks to the presence of the various modelers, projectors, and translators, each participant can express the models he or she is interested in, and use the expressive means that are more adequate to the

[1] Links are established and maintained in order to trace back the fragments to their source when needed.

problem at hand and to the participant's knowledge and habits. To assure that this requirement is satisfied three conditions must be met:

(a) the support system must include a modeler for the model to be expressed (and for any lower-level model upon which it is based);

(b) the support system must include a suitable projector, accepting as input the desired representation, and producing as output the corresponding model;

(c) the interaction environment must allow the participant to edit the chosen representation, and in particular it must include a suitable translator for the kind of media used to express the model.

2. **storing and retrieving models.** In our architecture, the semantic content of models is stored in the centralized semantic repository, so that it can be used by other modelers. The syntactic and pragmatic contents are stored in specialized syntactic repositories, while the lexical content is present in both repositories.

   non appartengono infatti, può testuale), ma (un impressione è che gli aspetti sintattici andrebbero alla coppia (modello, rappresentazione), ma questo difficile da gestire. Al momento questo punto non è perché abbiamo una sola rappresentazione per ogni con una sola eccezione costituita dal lemmario frequenza. Ad ogni modo, la nostra unica editabile è quella NL, e di questa struttura sintattica. Da

3. **model synthesis.** The opportunity for a modeler to synthesize a model is ensured, in our architecture, by its ability to access both the semantic repository data and the models built by other modelers. There are no architectural constraints on the representations that can be obtained, as long as

(a) the support system includes a modeler for the desired model (and for any lower-level model upon which it is based);

(b) the support system includes a suitable projector, accepting as input the given model, and producing as output the desired representation;

(c) the interaction environment is capable of visualizing the chosen representation, and in particular it includes a suitable translator for the media needed by the model;

(d) the semantic repository contains enough information to synthesize the desired model.

4. **preservation of lexical and pragmatic aspects.** Since the relevant information is stored in the semantic and syntactic repositories, modelers can maintain the lexical content of the requirements, and projectors can reproduce or map the pragmatic aspects as appropriate.

5. **formal validation and verification of properties.** The verification of arbitrary properties (which includes the formal validation of models) can be seen as the synthesis of a higher-level model, built upon the model to verify by a special-purpose modeler.[2] The outcome of the verification/validation can be presented to the user by a standard projector. Obviously, the model to verify must be synthesizeable by the support system, thus the conditions outlined above still apply.

6. **extraction of metric data.** As in the case above, a set of metric data can be seen as a model computed by a special-purpose modeler. Consequently, considerations analogous to those made above hold.

7. **suitable representations.** This requirement is satisfied by construction, due to the presence of the projectors components. It is worthwhile to remark that different projectors can provide different representations of the same model, and that each user can customize its own interaction environment to better suit his or her habits.

8. **extendibility.** In our reference architecture, extendibility is assured by the network structure itself. A new model or representation can be integrated simply by adding needed modelers, projectors, and/or translators, provided that

   (a) either the semantic repository or lower-level modelers provide the data needed for the new model or representation;

   (b) in case of a new representation, the interaction environment can visualize (and, possibly, edit) the representation;

   (c) the network structure is (statically or dynamically) updated to reflect the presence of the new modules.

## 4.3 Architecture specification

In this section the five classes of components of Circe's modular architecture are introduced and described in formal terms. A few examples are also provided, and the characteristics of the interconnection structure among the components are outlined.

### 4.3.1 Repositories

Circe's semantic repository is a flat typed-tuples space, where each tuple has the form $\langle\ tid, \mathsf{type}, arg_1, \ldots, arg_n\ \rangle$. The `type` indicates the kind of semantic information

[2]This approach has additional advantages on compositionality grounds; for example, it is possible to envision an even higher-level modeler built on top of various validation modelers that prioritises all problems found according to some given criteria.

embodied by the tuple, and a set of arbitrary parameters specifies the semantic content. Usually, parameters are either terms from the lexical part of some requirement or references to other tuples, via their tuple identifier *tid*. In this way the repository, although stored as a flat set, can actually maintain information shaped in more complex structures (e.g., trees or graphs). The semantic repository can be queried via basic queries based on the usual regular expression syntax, or by using more sophisticated matching operators to select entire substructures. Figure 4.2(b) constitutes a simple example of the contents of our semantic repository.

In formally describing models, we will sometimes use a more compact notation. In particular, the existence of a tuple $\langle tid, \mathsf{type}, arg_1, \ldots, arg_n \rangle$ in a tuple space $\mathcal{T}$ may be written as

$$\mathsf{type}(arg_{a_1}, \ldots, arg_{a_p})_{\mathcal{T}} = \langle arg_{r_1}, \ldots, arg_{r_q} \rangle$$

as long as the following conditions hold:

1. $\langle tid, \mathsf{type}, arg_1, \ldots, arg_n \rangle \in \mathcal{T}$
2. $\forall i = 1..n, (\exists k \in [1,p] \text{ s.t. } a_k = i \quad \vee \quad \exists k \in [1,q] \text{ s.t. } r_k = i)$
3. $\nexists k \in [1,p], h \in [1,q] \text{ s.t. } a_k = r_h$
4. $(\langle tid', \mathsf{type}, arg'_1, \ldots, arg'_n \rangle \in \mathcal{T} \wedge \forall i = 1..p, arg'_{a_i} = arg_{a_i}) \Rightarrow$ $\forall j \in [1,q], arg'_{r_j} = arg_{r_j}$

In other words, we can treat tuples as the extensional definition of a function denoted by its $\mathsf{type}$, as long as the function is well-defined. In this case, some of the tuple's arguments take the role of parameters, while the remaining arguments take the role of value of the function. Notice that we drop the tuple identifier *tid* in this notation. For the sake of conciseness, we will usually omit the tuple space $\mathcal{T}$, when it can be inferred from the context. Moreover, when condition 4 does not hold, we treat the function as returning a set: formally,

$$\langle arg_{r_1}, \ldots, arg_{r_q} \rangle \in \mathsf{type}(arg_{a_1}, \ldots, arg_{a_p})_{\mathcal{T}}$$

where conditions 1-3 above hold. In most circumstances, the mapping $a_1 \ldots a_p$, $r_1 \ldots r_q$ will be left implicit, and described once when a particular model is introduced. We also write $\mathsf{type}(arg_1, \ldots, arg_n)$ as a shorthand for $\mathsf{type}(arg_1, \ldots, arg_n) \in \mathcal{T}$. As an example, consider the following tuple space $\mathcal{T}$:

$$\mathcal{T} = \{ \quad \langle \text{ 1,}\mathsf{type}\text{,a,b,c } \rangle, \quad \langle \text{ 2,}\mathsf{type}\text{,a,d,e } \rangle \quad \}$$

Depending on our mapping convention for $\mathsf{type}$, stated when $\mathsf{type}$ is introduced, we could write either

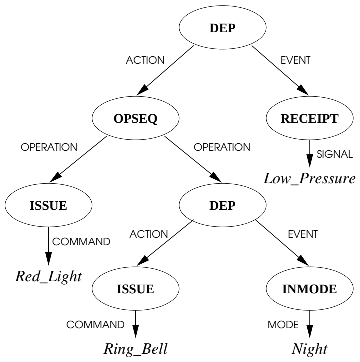
$$\mathsf{type}(a) = \{\langle b, c \rangle, \langle d, e \rangle\}$$

*Upon receipt of a Low Pressure warning, the system shall issue a Set Red Light command and, if in Night mode, issue a Ring Bell command.*

(a)

⟨ 1,issue,Ring_Bell ⟩
⟨ 2,issue,Red_Light ⟩
⟨ 3,inmode,Night ⟩
⟨ 4,dep,@1,@3 ⟩
⟨ 5,opseq,@2,@4 ⟩
⟨ 6,receipt,Low_Pressure ⟩
⟨ 7,dep,@5,@6 ⟩

(b)




(c)

Figure 4.2: A simple requirement (a), the corresponding tuples (b), and their structure (c).

or, by using the first two arguments to retrieve the third,

$$\begin{aligned} \mathsf{type}(a,b) &= c \\ \mathsf{type}(a,d) &= e \end{aligned}$$

Syntactic repositories are usually stored in the same format, but may also reference or contain data in arbitrary formats (e.g., styled text or video sequences), that are interpreted directly by the associated modeler.

In the current version of Circe, all repositories are kept under formal control (through the classic RCS system [111]). It is thus possible to trace back the entire evolution of the requirements, for analytical, statistical, or metric purposes. Process monitoring and control is performed in our architecture by considering these RCS repositories as syntactic repositories of specialized process metrics modelers.

### 4.3.2 Modelers

Circe's modelers are components that, given a set of input tuples, produce a set of output tuples representing a certain model (abstraction) of the system described by the input tuples. Certain modelers can also provide the opposite function, by fragmenting high-level models into more elementary tuples.

Each modeler is a triple $\langle input, output, behaviour\rangle$ where *input* declares which tuples are expected as input (by listing or giving a pattern of their types), *output* declares which tuples are produced as output (by listing their schemas), and *behaviour* describes the computation needed to produce the output tuples.[3]

As an example, Figure 4.3 shows a very simple modeler providing a "black box" model of a system (showing only the inputs and outputs of the system, while ab-

[3] Although the behaviour part can be arbitrary, we have found AWK scripts [1] to be a practical and reasonably efficient way of processing textually-encoded tuples. The system however provides means to invoke any executable as a modeler, so our particular choice is not limiting.

| **BlackBox** modeler | |
|---|---|
| `#INPUT: (issue\|receipt)` | |
| `#OUTPUT: hasin signal*` | |
| `#OUTPUT: hasout command*` | |
| `BEGIN` | `{ in=""; out=""; }` |
| `issue` | `{ out=out $3 }` |
| `receipt` | `{ in=in $3 }` |
| `END` | `{ outtuple("hasin",in); outtuple("hasout",out); }` |

⟨ 8,hasin,Low_Pressure ⟩
⟨ 9,hasout,Ring_Bell,Red_Light ⟩

Figure 4.3: The BlackBox modeler (left) and the model it computes on the repository of Figure 4.2(b) (right).

stracting from its behaviour). The model computed by this modeler from the single requirement in Figure 4.2(a) is also shown.

Formally, we describe a modeler $M$ as

$$M = \langle i, o, f \rangle$$

where $i$ is a set of tuple types[4], $o$ is a finite set of tuple types and $f$ is a function from tuple spaces to tuple spaces such that

$$\forall \mathcal{T}, \mathcal{T} \subseteq f(\mathcal{T})$$

(i.e., $f$ does not consume tuples in its input set, but may add new tuples). This condition on $f$ is needed to ensure non-retractability of requirements (and also to ensure the correctness of the linearization process described in Section 4.5.1).

Circe currently includes over 50 modelers, ranging from traditional models like entity-relationship and data flow diagrams, to more specialized ones like GUI, document structure, and timed message charts. Also included are several validation modelers (e.g., DFD well-formedness) and some metrication modeler (e.g., function points metrics). Some of these modelers are described in Chapter 6.

### 4.3.3 Projectors

Circe currently includes almost 30 projectors, that are in charge of building effective representations of the abstract models produced by the modelers. Projectors for editable representations (called *inbound projectors*) also provide parsing capabilities for a given representation. A projector is actually just a slight variation of a modeler, the only difference being that a projector outputs a (suitably encoded) concrete representation of a model, while a modeler outputs a set of tuples describing the

[4] As said above, this set can be described in the declaration of a modeler by a set of patterns (regular expressions). We will often treat patterns and the sets they represent interchangeably; where needed, we will use $i$ to denote the set and $\hat{\imath}$ to denote the pattern.

model. Moreover, a projector specifies which translator should handle its output. Thus, a projector is a quadruple $\langle input, output, behaviour, translator\rangle$ where *input* and *behaviour* are the same as for modelers, while *output* declares the symbolic name of the representation (for user display purposes) and *translator* is the symbolic names of a translator.

More formally, a projector $P$ is described as

$$P = \langle i, o, f, t\rangle$$

where $i$ is a set of tuple types (as for modelers), $o$ is the symbolic name for a view, $f$ is a translation function from tuple spaces to a language $\mathcal{L}_t$, where $t$ is the symbolic name for a translator, producing the description of a view expressed in language $\mathcal{L}_t$ from an input model. A sample projector for our BlackBox model is shown in Figure 4.4.

A projector that deserves special mention is our natural language parser CICO [3, 5], that is described in more detail in Chapter 5. It can parse NL requirements according to a given set of rules, and produce tuples that are then injected in the semantic repository. In Figure 4.2, the translation from the NL text in (a) and the tuple space in (b) is an actual example of CICO's output.

### 4.3.4 Translators

A translator is an interpreter for the language used by projectors to describe representations. As such, it is only characterized by the *symbolic name* for the language it can interpret (which usually coincides with the name of the translator) and by its *behaviour*. As an example, the current version of CIRCE includes translators for rich text, directed graphs, histograms, tables, and interactive forms for data entry.

Translators interface with the interaction environment is not specified by the architecture of the system; rather, it is dictated by the particular environment the translator is operating in.

**BlackBox** projector

```
#INPUT: (hasin|hasout)
#OUTPUT: "Black-box view"
#TRANSLATOR: graph
hasin   { for (i in $*)
              outcmd("arc",nil,"",i) }
hasout  { for (i in $*)
              outcmd("arc","",nil,i) }
```

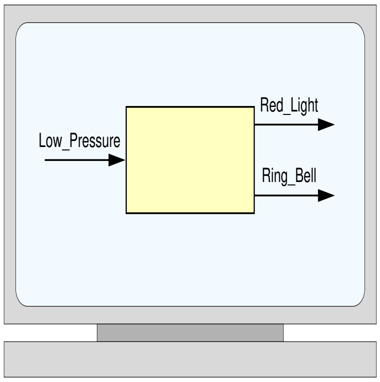



Figure 4.4: The BlackBox projector (left) and the representation it produces from the BlackBox model of Figure 4.3.

### 4.3.5 The interaction environment

Circe uses as its main interaction environment one of the environments that are most familiar to users, namely that of the World Wide Web. Representations output by the projectors are converted by the translators into HTML, Java and JavaScript and sent to the interaction environment through the HTTP protocol. This choice maximizes the availability of the interaction environment: a standard Web browser is enough to access and work profitably with Circe. This also simplifies the work of translators, that can rely on the layout capabilities of HTML and on the availability of standard Java or Javascript components to generate the respective representations.

Another advantage that has proven to be of paramount importance in practical use is the seamless integration of the various representations with widespread personal and group communication methods: users of the system can exchange the representations of their models via e-mail or internal discussion groups, and can easily publish snapshots or even “live” diagrams on the Web without leaving the interaction environment. This integration greatly simplifies direct interaction among the stakeholders, and this in turn favours conflict resolution and composition.

Currently, four types of representations are supported, corresponding to the four translators introduced above: read-only or editable text, read-only or editable graphs, histograms, and textual tables. Different views can communicate among themselves via a common message broker implemented in Javascript, or with the server (that is, with the translator that can parse their representation) via the HTTP POST mechanism, either via HTML FORMs, or by direct interaction in Java.

The message broker is used by the views to export and invoke functionality. For example, a validation view signaling an error in a certain requirement can wrap the requirement number in a link that, when pressed, sends a message to all other open views, asking them to highlight the questioned requirement. Local interactivity among views like the one described provides support for efficient navigation among different representations of the requirements.

### 4.3.6 The interconnection structure

The interconnection *mechanism* in Circe is quite simple, as it is based on the standard pipe mechanism for interprocess communication. The interconnection *structure*, on the other hand, is defined dynamically for each computation by a scheduler inside Circe’s kernel (see Section 4.5).

Upon request of a certain view, the scheduler searches Circe’s library of projectors to identify a projector that can produce the requested view. From the projector’s declaration, the *output* part is used to select the translator appropriate for the projector’s output language, while the *input* part is used to access the modelers library in order to find all modelers that can provide the needed tuples (according to their *output* declarations). The process is then repeated to satisfy those modelers *input* declarations, recursively propagating dependencies until a structured set of modelers that can compute the required model from the content of the semantic repository is found. All the modelers, the projector, and the translator are then executed in parallel, after having set up the interconnection structure according to their dependencies. The dual process is performed to parse editable representations and feed the semantic repository.

The advantages of a dynamic interconnection structure are twofold. First, it assures that only relevant modelers and translators are called upon for each computation, thus fostering efficiency. Second, and more important, it permits almost unlimited extendibility: whenever a new modeler, projector, or translator is added to CIRCE's library, it can be immediately used, without any need to adapt configuration or change pre-existing components. The strict separation of interface and implementation adopted in every CIRCE component (by separating the *input/output* and *behaviour* parts), and the strict encapsulation of all of them (modules can only access tuples, never functionality), are both consistent with this goal.

## 4.4 Reflection and dynamics

Among the syntactic repositories that are accessed by modelers, one is particularly important, and deserves a special treatment. This special repository is CIRCE itself, or more exactly, the static representation of the running system.

Modelers can access the static representation of any system component (repositories, modelers, projectors, and translators) and other data like user account information[5], parse rules, system glossaries, and maintenance data.

This makes the system highly reflexive. A user with appropriate authority can inspect or even change aspects of the running system through models and views in exactly the same way he or she can operate on the system described by a requirements document. Moreover, changes are immediately effective on the system without any need for a restart. As an example of a reflexive view, Figure 4.5 shows the actual architecture of CIRCE, in terms of dependencies among modules, produced by CIRCE itself via a reflexive modeler. The dependency graph is too intertwined to be traced effectively in such a representation[6], but in the leftmost part of the window the dependency chain leading from the *MASRules* and *ReflexDeps* modelers, through the *ReflexArchi* projector and finally to the *graph* translator is visible. This is exactly the computation that produced the view shown.

Other reflexive views available in the current version of CIRCE include parse rules, tuples signatures and programmer's documentation for each module, extracted directly from the sources of the modules.

[5]With the obvious provisions for security, e.g., encryption of passwords.

[6]Filtering mechanisms are provided in the user interface to only display portions of a complex graph.

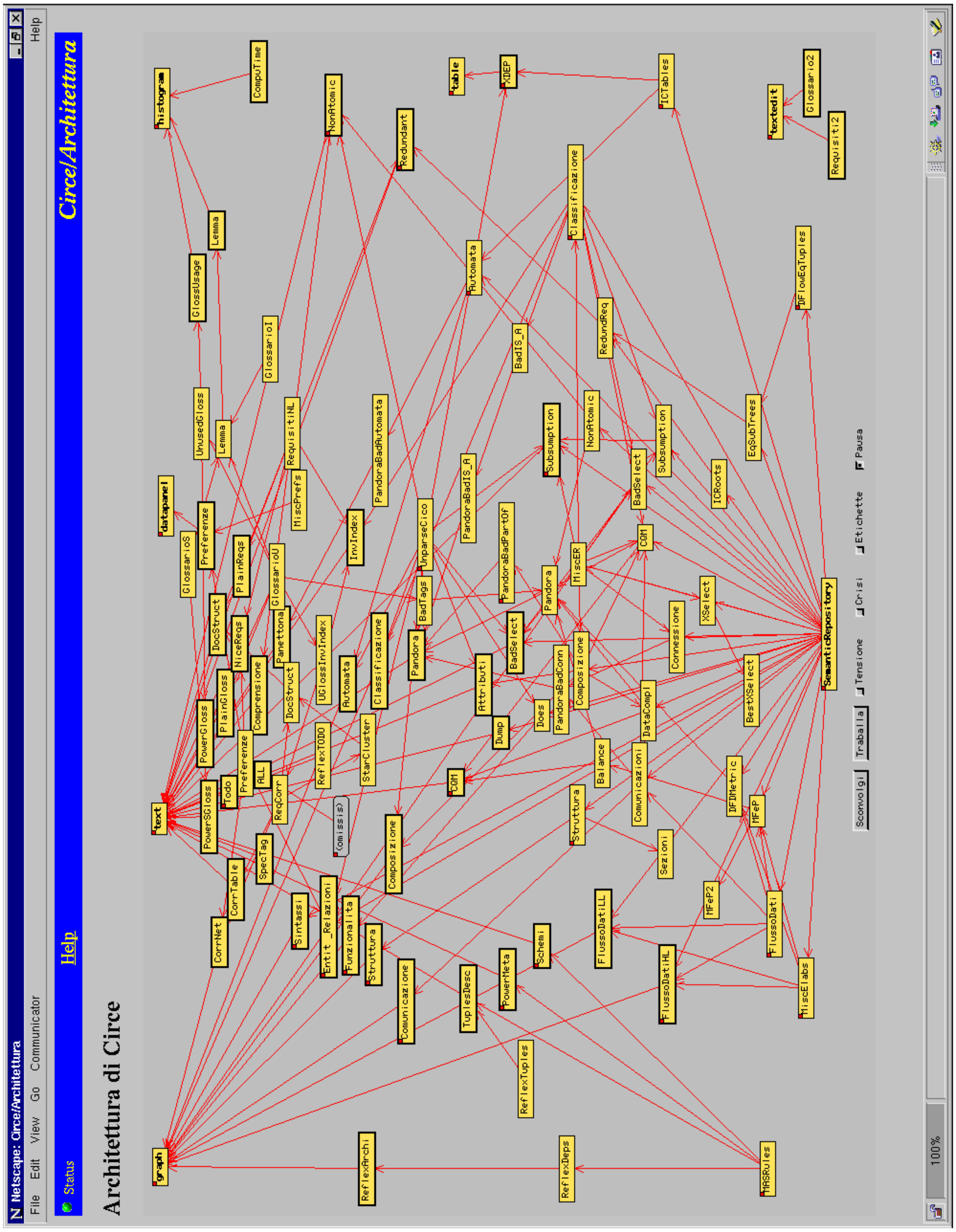



Figure 4.5: A reflexive view on Circe's architecture.

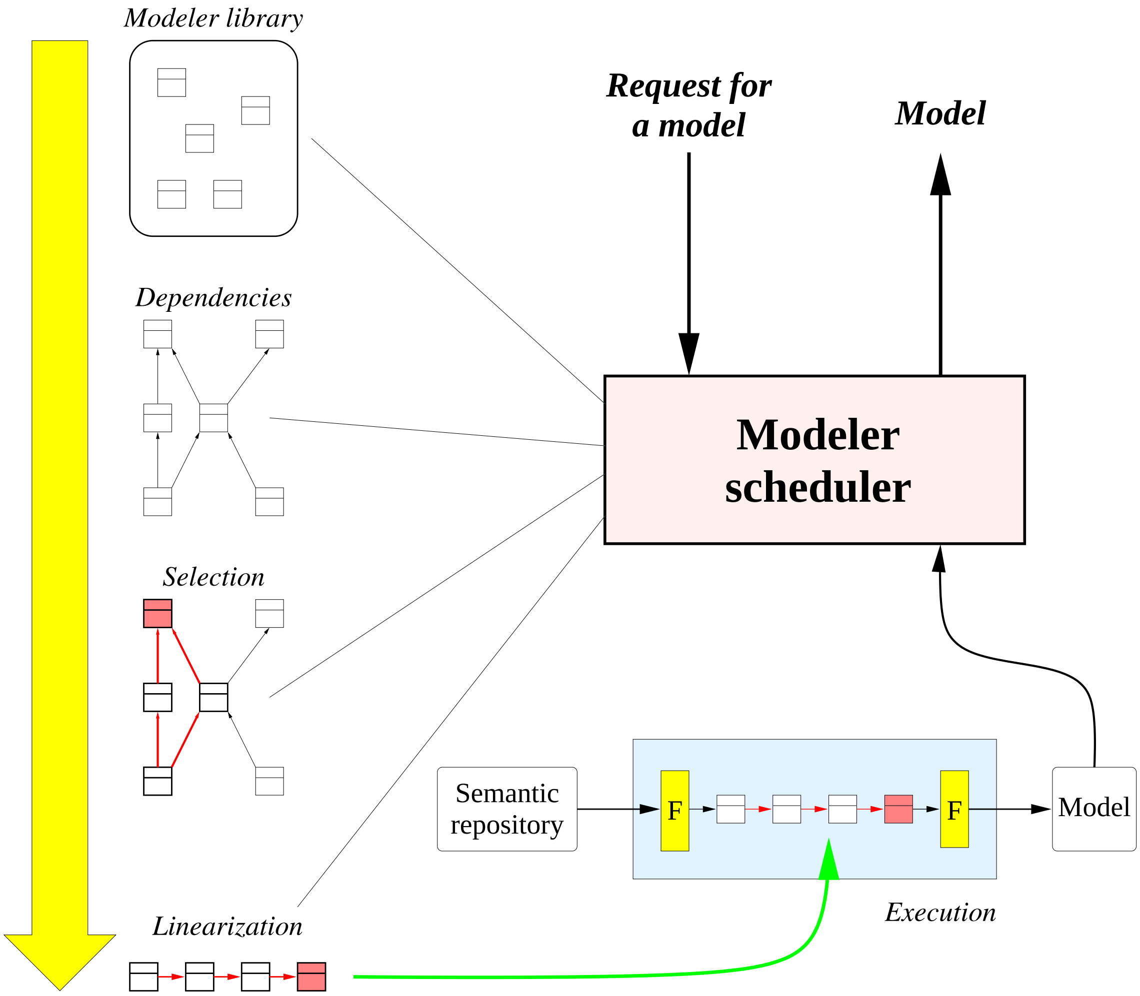



Figure 4.6: Model scheduling by the kernel.

## 4.5 The Kernel

In designing the system, an effort was made to keep the embedded functionality, i.e., the kernel component, as compact as possible. All knowledge concerned with specific domains, requirements, models, documents and media was moved into the corresponding class of modules, and the kernel was left with only the responsibility of appropriately scheduling the different modules to satisfy legitimate requests by the users.

### 4.5.1 Model scheduling

In CIRCE, a *model* is just a set of tuples that, under some specific interpretation, describes an abstraction of a system. As far as the kernel is concerned, however, tuples are never interpreted. While some of the tuples constituting a model can be found in the semantic repository, in most cases such tuples are synthesized by a hierarchy of modelers, as described in Section 4.1. One of the main functions of the kernel is thus to organize a computation by scheduling the modelers in such a way that all the tuples requested are correctly synthesized. This function is called *model scheduling*, and a part of the kernel called the *modeler scheduler* is responsible for it.

Figure 4.6 gives an overview of the process. The only input to the model scheduler is a *request for a model*, expressed as a pattern of tuple types, while the output is a set of tuples describing the model. When requested a model, the modeler scheduler scans the modeler library to build a *dependency graph*.

Formally, given a set of modelers $\mathcal{M}$, the associated dependency graph is

$$\mathcal{DG} = (\mathcal{M}, E)$$

where, given $m_1 = (i_1, o_1, f_1) \in \mathcal{M}$ and $m_2 = (i_2, o_2, f_2) \in \mathcal{M}$,

$$(m_1, m_2) \in E \quad \stackrel{def}{\Longleftrightarrow} \quad i_1 \cap o_2 \neq \emptyset$$

Since in the declaration of $m_1$, $i_1$ is expressed via a pattern $\widehat{i_1}$, the modeler scheduler actually checks the equivalent property

$$(m_1, m_2) \in E \quad \stackrel{def}{\Longleftrightarrow} \quad \exists q \in o_2 \text{ s.t. } q \sim \widehat{i_1}$$

where $\sim$ is the regular expression matching operator.

The model scheduler then selects the *heads* in $\mathcal{DG}$ for the given request $Q$. The heads are those modelers that directly produce the tuples requested by $Q$; formally $m_h = (i_h, o_h, f_h)$ is a head for $Q$ if $o_h \cap Q \neq \emptyset$ or, in concrete terms,

$$H = \{m_h \mid \exists q \in o_h \text{ s.t. } q \sim \widehat{Q}\}$$

Then the set of all modelers on which the heads depend (the *active set* $A$) is computed as follows:

$$A = H \cup \{m \in \mathcal{M} \mid \exists q \in A \text{ s.t. } (q, m) \in E\}$$

In a well-formed Circe system, $\mathcal{DG}$ is a directed acyclic graph,[7] thus it induces a partial order on $A$. The model scheduler *linearizes* the active set by selecting a total order compatible with the partial order established by $\mathcal{DG}$, and uses the linearization to schedule a computation by composition of the behaviours. Formally, if $\langle i_1, \ldots, i_n \rangle$ is the total order selected for modelers in $A$, the composition scheduled is

$$F(s) = f_{i_n}(f_{i_{n-1}}(\cdots f_{i_1}(s)))$$

[7] This is a design decision, motivated by efficiency considerations; this property is also checked by one of the reflexive validation modelers. If needed, fixed points can still be computed by confining the computation inside the behaviour part of some modeler.

Finally, the modelers are run on the semantic repository $\mathcal{S}$, filtered by an optional input filter $F_I$, to produce the set of tuples requested. The results can be filtered by an optional output filter $F_O$ to obtain a superset $M^*$ of the model $M$, under the filters in use (remember that the result of a modeler is a superset of its input, according to the definition of modeler given in Section 4.3.2). Formally,

$$M^* = F_O(F(F_I(\mathcal{S})))$$

where it is assumed that filters, if not used, are substituted by the identity function. Finally, $M^*$ is filtered through $Q$ to obtain the requested model $M$:

$$M = \{q \in M^* \mid q \sim Q\}$$

Input and output filters can be set by the user by selecting among a small library of parametric filters offered by the interaction environment. Input filters are used to restrict the contents of the semantic repositories to tuples satisfying certain conditions. This can be useful to observe, validate, or measure models on only part of a system. Currently, filters are provided to select only tuples deriving from certain requirements (e.g., to selectively include or exclude certain sections of a requirements document), or tuples of certain types (e.g., to restrict the analysis to tuples describing data flows), or tuples in which certain terms occur among the arguments (e.g., only tuples related to a certain subsystem). Output filters are used to reduce the amount of information presented to the user, thus focusing on certain characteristics. Filters are provided based on the type and contents of the tuples.

Notice that input filters restrict what information is available to modelers, so that the information filtered out is completely ignored in all the subsequent computations, while output filters only restrict the results of the computation, so that information filtered out is ignored by projectors and translators, but not by other modelers.

When a request for a model is presented, the modelers library is scanned, and each modeler is analysed to extract its input and output declarations. The information extracted is used to build the dependency graph $\mathcal{DG}$ and to identify the heads in $H$. Once the active set and a suitable linearization are found by visiting the graph, all the modelers and the optional filters, if present, are run in a pipeline having as source the current semantic repository. The output of the pipe is finally filtered through the request $Q$ to yield the desired model.

Executing all the modelers and filters in a pipeline has the added benefit of providing a natural way to distribute computational load in a multiprocessor machine. While no formal assessment has been conducted on this aspect, we expect that Circe would substantially benefit from a parallel architecture with almost-linear speedup in the range from 2 to 8 processors.

### 4.5.2 View scheduling

The process of obtaining a concrete representation of a view requested by the user is called *view scheduling*, and is illustrated in Figure 4.7. The component responsible

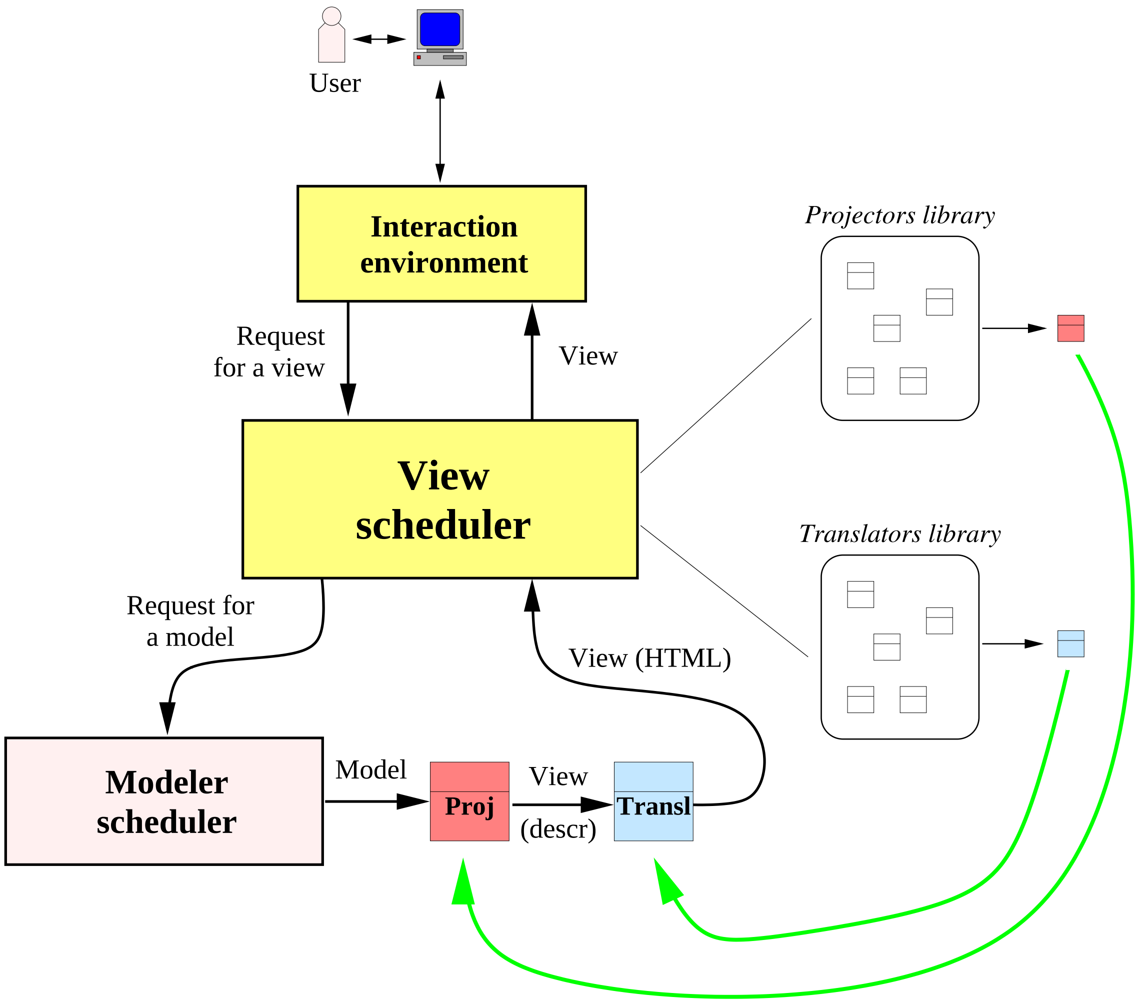



Figure 4.7: View scheduling by the kernel.

for this process is the *view scheduler*. It receives the symbolic name of a view, requested through appropriate means in the interaction environment by the user, and sets up a computation to produce the view requested.

The first step the view scheduler performs is the identification of the set $P$ of suitable projectors ($P$ is a subset of the set of all projectors in the system $\mathcal{P}$), i.e., of all projectors offering the requested view $v$:

$$P = \{p \in \mathcal{P} \mid p = (i, o, f, t) \wedge o = v\}$$

If $\#P > 1$ (i.e., more than one projector offers the requested view) the user is offered a choice, through the interaction environment, among the possible projectors. In the most common case, only one projector will be found offering a given view, and the user is not prompted for a choice.

Let $\bar{p} = (\bar{i}, \bar{o}, \bar{f}_p, \bar{t}\,)$ be the selected projector. The view scheduler uses $\bar{t}$ to select the translator $t = (\bar{t}, f_t)$ needed by $\bar{p}$ from the translators library, and invokes the model scheduler requesting the model described by $\bar{i}$. A computation is then set up to produce the concrete view $w$ corresponding to the requested view $v$:

$$w = f_t(\bar{f}_p(\mathsf{Model}(\bar{i})))$$

where $\mathsf{Model}(r)$ represents the invocation of the modeler scheduler with request $r$.

The projector $p$ and the translator $t$ are run in a pipeline having as input the model output by the model scheduler. The output of this computation is a concrete rendering for the view $v$ (i.e., with the Web-based interaction environment, an HTML page showing the view).

### 4.5.3 User authentications and authorization

Circe uses the identification scheme provided by HTTP basic authentication [46]. Basic authentication is not considered a secure scheme (as the user name and password are passed over the network as plain text), but if used in conjunction with a secure communications channel between the interaction environment and the server side of the system, e.g., SSL [30], it offers reasonable security.

Internally, each (*user*, *password*) pair is associated with a *group* and with a *project*. Moreover, Circe tracks the name and address of the *host* from which the user is connecting to the system. This information is used to determine the repositories and the views to which a user of the system has access.

In particular, the user's repository is uniquely identified by the (*user*, *group*, *project*) triple; it is also assumed that a user has read access to the repositories of other users in any *group* working on the same *project*.

The authority to access a certain view, either in read-only mode or, when applicable, in write mode, is determined by Circe's administrator.

# Chapter 5

# Cico: a simple NL parser based on fuzzy matching

Abstract

Cico is a simple parser for natural language statements. It is particularly suited for parsing of natural language requirements, as in this case several simplifying assumptions hold, and as such it is used as an inbound projector in Circe. However, it can also be used (and has been used) as a general domain-based parser. In this chapter, the parsing algorithm implemented by Cico is described, and some experimental data on its performance are given.

## 5.1 Parsing for NL requirements

Many commercially available tools offer a good support to techniques like tracing and evolution of requirements, linking requirements to other documents, structuring, merging, and splitting requirements document, assigning authorities over requirements to different individuals, etc. All these techniques, however, treat natural language requirements as black-box objects, without getting to their meaning. On the contrary, we are interested in extracting the semantic content from natural language requirements, in order to use the knowledge extracted to build, validate and measure models in our environment. To this end, parsing techniques for natural language requirements are needed.

Cico is a shallow parser for natural language requirements implementing domain-based parsing [5]. It can also be described as an engine for the application of a fuzzy rewriting system using backtracking and heuristic optimization strategies to determine an optimal parse tree for a statement.

While a long and honourable tradition of studies in the field of natural language processing exists (see [24] for a survey of the field), we had a particular set of conditions to comply with, and several simplification to take advantage of (both deriving

| NLP for requirements | NLP for arbitrary text |
|---|---|
| **Explicitness** is desired | Elliptical, depends on context |
| Mostly **interactive** redaction | Often batch processing |
| **Restricted** domain | Arbitrary domain |
| **Reduced** vocabulary | Vast vocabulary |
| **Well-defined** terms | Mostly approximate definitions |

Table 5.1: Peculiarities of NLP for requirements.

from the peculiar characteristics of requirements and of requirements processes) that suggested the use of non-standard techniques. Table 5.1 summarizes some of the peculiarities that make parsing of natural language requirements simpler than parsing of general text. These simplifications come mainly from three factors:

- **Explicitness**. It is useful to reduce to a minimum the amount of unexpressed information in the requirements. What one could consider common sense could be a total surprise to someone else, especially when they come from different cultural backgrounds. Since our target is not to obtain the most likely interpretation of incomplete or vague requirements, but rather to force the expression of requirements as explicit as possible (to minimize the probability of misunderstandings), an application geared to requirement engineering could dispense with huge knowledge bases like the Cyc ontology [73, 74] or ACAPULCO [53].

- **Interactivity**. Requirement writing is an interactive process. The user can repeatedly modify a requirement until it can be parsed by a tool; in our experience, this effort always makes the requirement clearer and often exposes real problems in the original version. Given the interactive nature of the drafting process, a failure in parsing a requirement is not fatal, differently from what happens in the case of non-interactive processes (when there is no way to obtain a different formulation of the text).

- **Repetitiveness**. The application domain, while largely variable, usually falls in a small number of well-know paradigms (reactive systems, data bases, information systems. . . ). Each of these paradigms can be analyzed independently, and a great part of the experience coming from one such project can be easily ported to a different project using the same paradigm. As a consequence, domain knowledge can be used profitably in the parsing process, thus obtaining better parsing results with increased performance.

The characteristics of our environment, however, establish some constraint on the parsing process. For example, the goal of independence from a particular syntactic form suggests that the dependency of the parsing process from the grammar rules of a given natural language should not be too tight. Moreover, the same interactivity aspect that makes parsing failures non-fatal, also imposes constraints on the amount of time that can be spent in parsing a requirement.

Cico tries to satisfy these constraints, and to take advantage of the peculiarities in Table 5.1, by adopting a shallow parsing approach based on fuzzy matching of domain-specific rules to natural language statements. In particular:

- Shallow parsing is used since explicitness promotes the adherence of the surface structure of a statement to its semantic content. Shallow parsing also fosters efficiency.

- Fuzzy matching is used to improve the robustness of the parsing technique, that can succeed even in the presence of wrongly ordered or missing items in a statement. This allows the user of the system to benefit from the support offered by the environment, even when starting from incomplete or badly structured textual material. Imperfect matching is acceptable thanks to the interactive nature of the environment, since the results of the parsing process are presented back to the user for confirmation.

- Domain-specific rules are used to encourage the use of standard language and style, and to simplify the association of a meaning to a certain syntactic form. Domain-specific parsing improves efficiency, since domain-specific rules tend to combine much less freely than grammatical rules, and make translation in different languages easier, since a map can be established between domain-specific rules on the same domain, even if in different languages.

In Circe, the role of Cico is that of an inbound projector: it parses natural language requirements documents input by the users, and populates the system's semantic repository with the resulting parse trees (encoded as a set of tuples). While our parsing technique supports both grammar-based tagging and domain-based tagging of language terms, alone or in combination, the rule sets currently in use only rely on domain-based tagging, thus simplifying the adaption to different natural languages.

## 5.2 Input format

Cico expects from its input stream a sequence of *free form statements*, obeying only a few lexical rules. In particular, it is assumed that the hash mark `#` introduces comments (extending to the end of the line), that a dot character `.` represents a full stop, and acts as a statement terminator, and that the single dash character `-`, when it appears in the first column, acts as an itemization marker for lists. Apart from that, valid text is essentially free-format, e.g.,

```
# Alarm handling.
```

```
# see DOD-STD such-and-such; last updated 22/3/99 BZ

The system can be in manual mode or in automatic mode. When the
alarm is active, if
- the system is in manual mode, or
- current time is greater than the last timestamp in the log,
the log daemon appends an entry for the alarm in the system log.
```

Before free text can be analyzed by the parser proper, however, a certain amount of normalization has to be performed in the text. This includes typographical adjustments, format handling, and glossary substitution, in order to transform free-form text into a sequence of *canonic statements*.

### 5.2.1 Typographical adjustments

Many human languages tend to have different typographical representations for certain alphabetic characters or punctuation signs. This is particularly relevant for electronic texts, that are often constrained by the historical limitations of the ASCII or ISO character sets.

For example, the German ß character can also be written `ss`; accented vowels in Italian can be written with an appended apostrophe, as in `e'` instead of `è`, quotation in many languages can be denoted by double quotes (`"..."`) but also by single or double backtick and apostrophe (`` `...' ``), or by guillemots (≪ ... ≫) — that can be themselves represented as << and >> — and so on.

To handle these and other similar language-specific issues, Cico performs a typographical canonization step on the source text. For the Italian language, this step includes:

- substituting proper ISO-8859-1 accented vowels for *vowel*+`'` sequences, choosing acute or grave accents where appropriate (e.g., `e'` → `è`, but `poiche'` → `poiché`), and taking care of exceptions consisting in truncated words (e.g., `po'` is a truncated form of `poco` and thus must be left intact);
- converting decimal points (e.g., `2.1`) into decimal commas (`2,1`);
- substituting various quotation styles with the uniform use of double quotes.

### 5.2.2 Format handling

The second canonization step is concerned with structural features of the text, and is totally language-independent. This step includes:

- reducing runs of multiple white space characters to a single space;
- discarding comment lines;

- discarding blank lines;
- joining lines and splitting them on full stops, so that each statement consists of a single line of text;
- substituting itemization markers with special markup to identify item and list contexts (see Section 5.3.8);
- making a single term out of quoted excerpts;
- discarding remaining unneeded punctuation characters.

At the end of this stage, the input stream consists of a sequence of requirements, one per line; each requirement consists of a sequence of words and special markers, separated by spaces. Formally, a requirement document $\mathcal{R}$ is defined as set of sequences of words:

$$\mathcal{R} = \{\langle \omega_1, \omega_2, \ldots, \omega_n \rangle\}$$

### 5.2.3 Glossary substitution

A glossary serves two purposes: it declares properties of the terms it contains, by annotating each term with a set of *tags*, and establish a map to the canonical form of a term from a set of synonyms. Formally, a glossary $\mathcal{G}$ is defined as a set of term-synonyms pairs:

$$\mathcal{G} = \{(\tau, \{\tau_i\})\}$$

where $\tau$ is the base term (i.e., the canonical form for the term) and $\tau_i$ are synonyms for $\tau$. The syntax for a glossary is given by the following context-free grammar

$$\begin{array}{rcl}
\text{GLOSS} & ::= & \epsilon \quad | \quad \text{ENTRY GLOSS} \\
\text{ENTRY} & ::= & \text{term TAGS SYNS .} \\
\text{TAGS} & ::= & \epsilon \quad | \quad \texttt{/}\ \text{tag TAGS} \\
\text{SYNS} & ::= & \epsilon \quad | \quad \texttt{:}\ \text{SYNL} \\
\text{SYNL} & ::= & \text{term} \quad | \quad \text{term , SYNL}
\end{array}$$

Typical glossary entries for the example given above are

```
last timestamp/ATTR: time of the last update.
system log/INF: log, system journal, logbook.
system/HASMODE/SYS.
manual mode/MODE.
automatic mode/MODE: unsupervised.
log daemon/ELAB.
```

Given a term $\tau$, we define the two functions $\mathsf{term}(\tau)$ and $\mathsf{tags}(\tau)$ to be, respectively, the base form of the term (denoted by "term" in the syntactic rule for ENTRY given above) and the set of its tags (denoted by TAGS in the same syntactic rule).

Glossary substitution is obtained by applying the $\mathsf{map}_{\mathcal{G}}()$ function described below to each $\omega_i$ in each requirement in $\mathcal{R}$:

$$\mathsf{map}_{\mathcal{G}}(\omega) = \begin{cases} \tau & \text{if } \exists(\tau,\sigma) \in \mathcal{G} \text{ s.t. } \omega \in \sigma \ \vee \ \omega = \mathsf{term}(\tau) \\ \omega & \text{otherwise} \end{cases}$$

To guarantee uniqueness in the definition above, we assume that glossaries are always *well-formed*, i.e., there is at most one base term for each possible word. Formally, $\mathcal{G}$ is well-formed if the two conditions below hold:

$$\forall(\tau,\sigma) \in \mathcal{G}, \forall\tau_i \in \sigma, \nexists(\tau',\sigma') \in \mathcal{G} \text{ s.t. } (\tau_i \in \sigma') \wedge (\tau' \neq \tau)$$

$$\forall(\tau,\sigma) \in \mathcal{G}, \forall(\tau',\sigma') \in \mathcal{G}, \mathsf{term}(\tau) = \mathsf{term}(\tau') \Rightarrow \tau = \tau'$$

Given a statement $s = \langle\omega_1,\ldots,\omega_n\rangle$ we denote its glossary-substituted version with $s_{|\mathcal{G}}$:

$$s_{|\mathcal{G}} = \langle\mathsf{map}_{\mathcal{G}}(\omega_1),\ldots,\mathsf{map}_{\mathcal{G}}(\omega_n)\rangle$$

## 5.3 Parsing algorithm

Cico uses a parsing algorithm based on the fuzzy matching of sentence fragments to templates contained in a set of rules. Since generally several matchings are possible at each stage, Cico performs an heuristically-limited search in the space of all valid parsing trees to determine the overall best parsing tree.

### 5.3.1 MAS rules

Cico's parsing algorithm is driven by a set of *MAS rules*. The MAS acronym stands for Model, Action, Substitution, and refers to the three parts in each rule. Formally, a MAS rule $\rho$ is defined as

$$\rho = (\mu,\alpha,\sigma)$$

where

$$\begin{aligned} \mu &= \langle\tau_1,\ldots,\tau_n\rangle \text{ is the } \textit{model}, \\ \alpha &= \langle\beta_1,\ldots,\beta_p\rangle \text{ is the } \textit{action}, \text{ and} \\ \sigma &= \langle\gamma_1,\ldots,\gamma_q\rangle \text{ is the } \textit{substitution}. \end{aligned}$$

The syntactic denotation for a MAS rule is

$$\tau_1 \ \ldots \ \tau_n \ :: \ \beta_1 \ \ldots \ \beta_p \ \texttt{>>} \ \gamma_1 \ \ldots \ \gamma_q$$

where the action or the substitution, but not both, can be empty (in the first case the :: separator can be omitted, as can the >> separator in the second case).

The intuitive meaning of a rule $\rho$ applied to a requirement $r$ is "if the model $\mu$ of $\rho$ matches a fragment of $r$, execute the action $\alpha$ and substitute the matching fragment with $\sigma$".

Terms in a rule are either constants or variables; the function $\mathsf{isvar}(\tau)$ is true if $\tau$ is a variable, and $\mathsf{isconst}(\tau)$ is true if $\tau$ is a constant (obviously, $\mathsf{isvar}(\tau) = \neg\mathsf{isconst}(\tau)$ always holds). We will use UPPER CASE to indicate constant terms and lower case to indicate variable names. When a rule is applied, the matching can bind a value to the variable terms in $\mu$. This binding is then used when the rule is fired to assign values to the variables in $\alpha$ and $\sigma$.

### 5.3.2 Matchings

Given a term $\tau$ and a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$, we define a set of possible *targets* for $\tau$ in $s$ as

$$\mathsf{target}(\tau, s) = \begin{cases} \{i \mid \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_i)\} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

A *matching map* between a rule $\rho = (\mu, \alpha, \sigma)$ (with $\mu = \langle \tau_1, \ldots, \tau_n \rangle$) and a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ is a sequence of targets for all the terms in the model of $\rho$, i.e.

$$\mathsf{match}(\rho, s) = \langle \mathsf{target}(\tau_1, s), \ldots, \mathsf{target}(\tau_n, s) \rangle$$

Given a matching map $M = \langle T_1, \ldots, T_n \rangle$, we say that $M$ is *complete* if $\forall i = 1..n, T_i \neq \emptyset$. A *valid match* $m$ from a complete matching map $M$ is defined as follows:

$$m = \langle t_1, \ldots, t_n \rangle \text{ s.t. } \forall i = 1..n, (t_i \in T_i) \wedge (\forall j = 1..n, t_i = t_j \Rightarrow i = j)$$

The intuition behind the above definition is that a valid match associates each element in the model of a rule to a single, distinct term in a statement. In most cases, a single complete matching map will contain many different valid matches. To choose among them, CICO uses a scoring system.

### 5.3.3 Match scoring

Since models express "natural" templates for sentence fragments, the ideal match for a model would be a one-to-one, ordered correspondence between elements of the templates and terms in a statement. However, natural languages usually permit a certain degree of freedom in the placement of words. This is particular true of the Italian language, that — as its ancestor, Latin — permits largely unrestricted rearrangement in the order of words.

Moreover, words not accounted for in the template could be added to a statement without altering the original meaning, but rather specifying a concept in greater detail. This is typical, for example, of adjectives and adverbs, that could well be discarded when analyzing a text to obtain at least a rough understanding of its meaning.

CICO associates a score to each valid match taking into account the two factors mentioned above. The latter is characterized as follows: given a match $m = \langle t_1, \ldots, t_n \rangle$, its *span* is given by the distance between the rightmost and leftmost target (inclusive), that is

$$\mathsf{span}(m) = \max_{i=1..n}(t_i) - \min_{i=1..n}(t_i) + 1$$

the number of unmatched terms in the statement fragment matched by $m$ is thus

$$U_m = \mathsf{span}(m) - n$$

With respect to ordering, we can simply consider the distance, expressed in number of terms, between two terms that occupy consecutive positions in the model as an indicator of bad ordering. For each target $t_i, i > 1$, we define the distance $D_i$ as

$$D_i = t_i - t_{i-1} - 1$$

(notice that this distance is 0 when $t_{i-1}$ and $t_i$ point to consecutive terms in the statement, and can be negative in case of an inversion).

Finally, the global matching score for a valid match $m = \langle t_1, \ldots, t_n \rangle$ is given by

$$\begin{aligned}\mathsf{mscore}(m) \quad = \quad & -K_{DROPPED} * U_m \\ & -K_{DIST} * \sum_{i=1..n|D_i>0} D_i \\ & -K_{INVERSION} * \sum_{i=1..n|D_i<0} -D_i\end{aligned}$$

where $K_{DROPPED}$, $K_{DIST}$ and $K_{INVERSION}$ are parameters of the cost model. Notice that $\mathsf{mscore}(m)$ is always less or equal to 0, with 0 assigned to the "perfect" match in which strict ordering is preserved and no term is dropped.

### 5.3.4 Rule firing

When CICO determines that a rule should be fired, it applies the bindings established by the match to the action and the substitution part of the rule, executes the action, and replaces the matched fragment in the original statement with the substitution.

Formally, given a term sequence $r$, a model $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, a statement $s = \langle \omega_1, \ldots, \omega_m \rangle$ and a valid match $m = \langle t_1, \ldots, t_n \rangle$ between $\mu$ and $s$, we define[1]

$$\mathsf{subst}(\mu, m, r) = r \left[{}^{\tau_1}\!/_{\omega_{t_1}}\right] \cdots \left[{}^{\tau_n}\!/_{\omega_{t_n}}\right]$$

[1]We use the notation $\alpha \left[{}^{b}\!/_{c}\right]$ to indicate the sequence $\alpha$, in which every instance of $b$ has been replaced by $c$.

The replacement of the fragment of $s$ matched by $m$ with the substitution $\sigma' = \langle \gamma'_1, \ldots, \gamma'_q \rangle$ is defined by

$$\mathsf{repl}(s, \sigma', m) = \langle \omega_1, \ldots, \omega_{min-1}, \gamma'_1, \ldots, \gamma'_q, \omega_{max+1}, \ldots, \omega_m \rangle$$

where

$$\begin{array}{rcl} max & = & \max_{i=1..n}(t_i) \\ min & = & \min_{i=1..n}(t_i) \end{array}$$

To characterize the application and firing of a rule $\rho = (\mu, \alpha, \sigma)$ to a statement $s$, we first define a *sa-structure* as a pair $(s, A)$ where $s$ is a statement and $A$ is a sequence of actions. The function[2]

$$\mathsf{apply}(\rho, m, (s, A)) = (\mathsf{repl}(s, \mathsf{subst}(\mu, m, \sigma), m), A.\mathsf{subst}(\mu, m, \alpha))$$

returns a sa-structure in which the substitution of $\rho$ has replaced the fragment of $s$ matched by $m$, and the action of $\rho$ has been appended to $A$.

### 5.3.5 Rule scoring

Beside the score due to the ordering of matches, a second score is computed depending on the *specificity* of a rule. Three factors influence specificity of a rule $\rho = (\mu, \alpha, \sigma)$ with $\mu = \langle \tau_1, \ldots, \tau_n \rangle$:

1. the length of the model, $ms = n$;
2. the total number of tags in the model, $ts = \sum_{i=1..n} \#\mathsf{tags}(\tau_i)$
3. the total number of constants in the model, $cs = \#\{\tau_i \mid \mathsf{isconst}(\tau_i)\}$

Moreover, we assign a fixed score to each rule to favour longer chains of rule applications (see Section 5.3.7). We define then

$$\begin{array}{rcl} \mathsf{rscore}(\rho) & = & ms * K_{TERMS} \\ & & +ts * K_{TAGS} \\ & & +cs * K_{CONSTS} \\ & & +K_{RULE} \end{array}$$

where $K_{TERMS}$, $K_{TAGS}$, $K_{CONSTS}$ and $K_{RULE}$ are parameters of the cost model.

[2] We use the notation $A.B$ to indicate the concatenation of the sequence $B$ at the end of $A$ (append).

### 5.3.6 Rests and rest scoring

A statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ that is not matched by any rule in a rule set $R$ is called a *rest* (with respect to $R$). The name refers to the fact that the basic parsing algorithm (see Section 5.3.7) repeatedly applies rules from $R$ to a statement until no rule can be applied. What is left is, indeed, a rest.

The trace of these repeated applications is called a *parsing sequence*. To encourage the selection of parsing sequences that leave as little unparsed material as possible, a score is given to each rest, computed as

$$\mathsf{lscore}(s) = -n * K_{REST}$$

where $K_{REST}$ is a parameter of the cost model.

### 5.3.7 Basic parsing algorithm

The basic parsing algorithm implemented by CICO can be described informally as follows:

1. given a statement in input, repeatedly apply all known rules, in turn, trying for each rule all possible valid matches, until no rule can be applied;
2. evaluate the score of each possible parsing sequence;
3. choose the best one and execute the relevant actions;
4. repeat from 1. until input exhausted.

This informal algorithm is described in more formal terms by Algorithms 1 and 2. Notice that Algorithm 1 selects internally the highest-ranking parsing sequence as the final result of the computation. This is used in CIRCE to ensure that each requirement is interpreted unambigously by other components of the environment (the same interpretation is also presented back to the user for confirmation). However, it is possible and sensible in different contexts to output *all* the possible parsing sequences, together with their score, leaving to other components — possibly with a better knowledge of the domain — the task of selecting the "right" parsing sequence for a given context.

### 5.3.8 Parsing contexts

As introduced in Section 5.2.2, the usage of lists in the source text can introduce distinct *parsing contexts* in a statement. We will use $\triangleleft$ and $\triangleright$ markers to indicate a parsing context, like in

$$\langle a, b, \triangleleft, c, \triangleleft, d, e, \triangleright, f, g, \triangleright, h, \triangleleft, i, j, \triangleright, k \rangle$$

The format canonization process inserts context delimiters around every item in a list, and around the entire list as a whole. Thus, a statement like

**Algorithm 1** BasicParse($s, R, A, k$).

{$s$ is a statement, $R$ is a set of MAS rules, $A$ is a sequence of actions and $k$ is a score}
$f := \mathsf{false}$
**for all** $\rho \in R$ **do**
  **if** $\mathsf{match}(\rho, s)$ is complete **then**
    **for all** $m$ valid matches from $\mathsf{match}(\rho, s)$ **do**
      $f := \mathsf{true}$
      $(s', A') := \mathsf{apply}(\rho, m, (s, A))$
      $k' := k + \mathsf{mscore}(m) + \mathsf{rscore}(\rho)$
      BasicParse($s', R, A', k'$)
    **end for**
  **end if**
**end for**
**if** $\neg f$ **then** {no rule applies, we are on a leaf node}
  $k' = k + \mathsf{lscore}(s)$
  **if** $k' > B_k$ **then**
    $B_k := k'; B_A := A; B_s := s$
  **end if**
**end if**
{$B_k$, $B_A$, and $B_s$ are respectively the score, the action sequence and the final rest of the best parse sequence, according to the scoring system}

**Algorithm 2** SimpleParse($\mathcal{R}, \mathcal{G}, R$).

{$\mathcal{R}$ is a set of statements, $\mathcal{G}$ is a glossary, and $R$ is a set of MAS rules}
**for all** $r \in \mathcal{R}$ **do**
  $B_k :=$ VeryBadScore
  BasicParse($r_{|\mathcal{G}}, R, \langle\rangle, 0$)
  **if** $B_k \geq$ MinimumAcceptableScore **then**
    execute $B_A$, final rest is $B_s$
  **else**
    final rest is $r_{|\mathcal{G}}$
  **end if**
**end for**

```
a b
- c d
- e
f g
```

becomes

$$\langle \text{a}, \text{b}, \overbrace{\triangleleft, \underbrace{\triangleleft, \text{c}, \text{d}, \triangleright}_{\text{first item}}, \underbrace{\triangleleft, \text{e}, \triangleright}_{\text{second item}}, \triangleright}^{\text{entire list}}, \text{f}, \text{g} \rangle$$

Language-specific optimizers can also insert context in other cases. For example, the optimizers for Italian and English establish contexts around parenthetical text (e.g., "The system shall set the alarm (initially: FALSE) to TRUE"), and around any text between an "if" and a "then" ("se" and "allora" for Italian).

Cico handles a parsing context by computing the best parsing sequence for the context in insulation with respect to the rest of the statement, executing the relevant actions and replacing the entire context with the resulting rest. The process is then iterated until no more contexts are left in the statement, at which time a parsing sequence for the entire statement is computed.

This process is described formally by Algorithms 3 and 4, where the functions introduced below are used.

Given a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ with embedded, correctly-nested context markers $\triangleleft$ and $\triangleright$, we define

$$\mathsf{hascontext}(s) = \exists i \in [1, n] \text{ s.t. } \omega_i = \text{“}\triangleleft\text{”}$$

A *context pointer* is a pair of indexes pointing to two matching context markers in $s$. In particular, we define the leftmost innermost context in $s$ by

$$\mathsf{firstcontext}(s) = (p, q)$$

where

$$\begin{aligned} p &= \max_{i=1..n} i \text{ s.t. } \omega_i = \text{“}\triangleleft\text{”} \;\wedge\; \forall h = 1..i-1, \omega_h \neq \text{“}\triangleright\text{”} \\ q &= \min_{j=p+1..n} j \text{ s.t. } \omega_j = \text{“}\triangleright\text{”} \end{aligned}$$

Given a statement $s$ as above, a context $c$ pointed by a context pointer $(p, q)$, and a second statement $r = \langle \zeta_1, \ldots, \zeta_m \rangle$, we also define the following two operations to extract $c$ from $s$ and to replace $c$ in $s$ by $r$:

$$\begin{aligned} \mathsf{extractcontext}(s, c) &= \langle \omega_{p+1}, \ldots, \omega_{q-1} \rangle \\ \mathsf{replacecontext}(s, c, r) &= \langle \omega_1, \ldots, \omega_{p-1}, \zeta_1, \ldots, \zeta_m, \omega_{q+1}, \ldots, \omega_n \rangle \end{aligned}$$

These operations are used in the full parsing algorithm discussed below.

### 5.3.9 Full parsing algorithm

The full parsing algorithm described by Algorithms 3 and 4 works by parsing deep contexts in insulation, accumulating resulting actions and substituting final rests in place of the original context.

**Algorithm 3** ContextParse($s, R$).

{$s$ is a statement, $R$ is a set of MAS rules}
**while** hascontext($s$) **do**
  $c :=$ firstcontext($s$)
  $r :=$ extractcontext($s, c$)
  $B_k :=$ VeryBadScore
  BasicParse($r, R, \langle\rangle, 0$)
  **if** $B_k \geq$ MinimumAcceptableScore **then**
    execute $B_A$
    $s :=$ replacecontext($s, c, B_s$)
  **else**
    $s :=$ replacecontext($s, c, r$)
  **end if**
**end while**
$B_k :=$ VeryBadScore
BasicParse($s, R, \langle\rangle, 0$)
**if** $B_k \geq$ MinimumAcceptableScore **then**
  execute $B_A$, final rest is $B_s$
**else**
  final rest is $s$
**end if**

**Algorithm 4** FullParse($\mathcal{R}, \mathcal{G}, R$).

{$\mathcal{R}$ is a set of statements, $\mathcal{G}$ is a glossary, and $R$ is a set of MAS rules}
**for all** $r \in \mathcal{R}$ **do**
  ContextParse($r_{|\mathcal{G}}, R$)
**end for**

It is worthwhile to notice that unparseable contexts are simply substituted back in the enclosing context, and that parsing is attempted again on the resulting block. Thus, generally context insertion does not change the resulting action sequence for a statement.

Although rare, however, there are cases in which the insertion of a context can change the parsing sequence. This can happen when multiple rules with the same score can be applied to the text inside a context, and a third rule acts as a discriminator when applied outside the context. For example, consider the rule set containing the following three rules:

$$\begin{aligned}
\rho_1 &= (\langle A, B\rangle, \alpha_1, \langle S\rangle) \\
\rho_2 &= (\langle A, B\rangle, \alpha_2, \langle T\rangle) \\
\rho_3 &= (\langle T, C\rangle, \alpha_3, \langle\rangle)
\end{aligned}$$

and the two statements, differing only in the presence of context markers around "A B",

$$\begin{aligned}
s_1 &= \langle A, B, C\rangle \\
s_2 &= \langle \triangleleft, A, B, \triangleright, C\rangle
\end{aligned}$$

The application of FullParse() to $s_1$ finds that the best parsing sequence is given by $\langle \rho_2, \rho_3\rangle$ (with the obvious matches), and causes the execution of $\alpha_2$ and $\alpha_3$, as one would expect. When applied to $s_2$, however, FullParse() would first call BasicParse() on $\langle A, B\rangle$, that could non-deterministically result either in the application of $\rho_1$, leading to a rest $\langle S, C\rangle$, or in the application of $\rho_2$, leading to $\langle T, C\rangle$. Both parsing sequences, in fact, have the same score. If the latter is chosen, parsing proceeds by firing $\rho_3$, thus causing the execution of $\alpha_2$ and $\alpha_3$, as in the previous case. But if the former is chosen, no rule matching $\langle S, C\rangle$ would be found, thus terminating with an action sequence $\langle \alpha_1\rangle$ and a rest $\langle S, C\rangle$.

In practical applications, this kind of ambiguity is extremely rare, and if present could be resolved by avoiding the use of context-introducing markup (e.g., lists).

## 5.4 Special forms of tagging

Three special forms of tagging exists, to support higher-level features of the Circe toolset. *Speculative tagging* is a means to leave the tag set for a term unspecified, and instruct the parser to compute a set of tags for the term that maximizes overall parsing score. Backward and forward *lookup references* support the implementation of pronouns, and specify that a term should be considered equivalent to a previous (resp. following) term in the statement, choosing the one that maximizes parsing score. Finally, *wildcard tagging* supports model terms that match any tagged term in the statement — regardless of the specific tags it has.

### 5.4.1 Speculative tagging

Speculative tagging is syntactically denoted by the special tag `/?`. Its effect is as follows: when testing for containment among tag sets in the definition of $\mathsf{target}()$ (see page 57), if `/?` $\in \mathsf{tags}(\omega)$ the test is always considered passed, and when the corresponding rule is applied, a special action is executed to record the tag set that is being speculatively associated with the given term. Formally, the definition of $\mathsf{target}()$ is changed to

$$\mathsf{target}(\tau, s) = \begin{cases} \{i \mid \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_i)\} \cup \{i \mid \mathsf{/?} \in \mathsf{tags}(\omega_i)\} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

Next, we introduce the special action $\alpha^?(\omega, T)$ whose purpose is to record that term $\omega$ is being speculatively assigned all the tags in the set $T$. Given a model $\mu = \langle \mu_1, \ldots, \mu_n \rangle$, a match $m = \langle t_1, \ldots, t_n \rangle$, and a statement $s = \langle \omega_1, \ldots, \omega_p \rangle$, we also define the *reverse match map*

$$\mathsf{rmap}(i, m) = \begin{cases} j & \text{if } \exists j \text{ s.t. } t_j = i \\ 0 & \text{otherwise} \end{cases}$$

and the auxiliary function

$$\mathsf{rst}(\mu, m, s, i) = \begin{cases} \langle \alpha^?(\omega_i, \mathsf{tags}(\mu_{\mathsf{rmap}(i,m)})) \rangle & \text{if } \mathsf{/?} \in \mathsf{tags}(\omega) \wedge \mathsf{rmap}(i, m) \geq 1 \\ \langle\rangle & \text{otherwise} \end{cases}$$

The definition of $\mathsf{apply}()$ thus becomes

$$\mathsf{apply}(\rho, m, (s, A)) = (\mathsf{repl}(s, \mathsf{subst}(\mu, m, \sigma), m), A.\mathsf{rec}_?(\mu, m, s).\mathsf{subst}(\mu, m, \alpha))$$

where

$$\mathsf{rec}_?(\mu, m, s) = \mathsf{rst}(\mu, m, s, 1). \cdots .\mathsf{rst}(\mu, m, s, p)$$

As an example of application of speculative tagging, consider the rule $\rho = (\langle x, y/\mathrm{TAG}_1, C \rangle, \langle \alpha_0 \rangle, \langle S \rangle)$[3] and the statement $s = \langle A, B/?, C \rangle$. By using the $\mathsf{target}()$ definition above, the matching function would yield

$$\begin{aligned} \mathsf{match}(\rho, s) &= \langle \mathsf{target}(x, s), \mathsf{target}(y/\mathrm{TAG}_1, s), \mathsf{target}(C, s) \rangle \\ &= \langle \{1, 2, 3\}, \{2\}, \{3\} \rangle \end{aligned}$$

This matching map is complete; of all the possible matches from the matching map, only the match $m = \langle 1, 2, 3 \rangle$ is valid (and has an acceptable score in the sense of algorithm 1), so the rule can be applied. In particular,

$$\mathsf{rec}_?(\langle x, y/\mathrm{TAG}_1, C \rangle, m, s) = \langle\rangle.\langle \alpha^?(B, \{\mathrm{TAG}_1\}) \rangle.\langle\rangle$$

so that the $\mathsf{apply}()$ function results in

$$\mathsf{apply}(\rho, m, (s, \langle\rangle)) = (\langle S \rangle, \langle \alpha^?(B, \{\mathrm{TAG}_1\}), \alpha_0 \rangle)$$

as expected. In other words, given the glossary declaration `B/?` meaning "what are the best tags for `B`?", the parsing process provides the answer "`B/TAG`$_1$ is the best declaration", where 'best' is always meant with respect to the scoring system and to the rule set in use, for the purpose of maximizing parsing score.

[3] Recall that lower case indicates variable terms, while upper case indicates constant terms.

### 5.4.2 Lookup references

Lookup references, denoted by `/<` (resp. `/>`), let a term inherit the properties (tags) of another term to its left (resp. right) in a statement. Whereas speculative tagging permits a term to assume *any* tag if that improves parsing score, lookup references only permit a term to assume the tags of another term in the same statement. Moreover, the assignment is not recorded as a special action, but rather the referred term is directly substituted for the reference in the action and in the substitution of the rule. Formally, the definition of $\mathsf{target}()$ is extended again to account for backward (`/<`) and forward (`/>`) references:

$$\mathsf{target}(\tau, s) = \begin{cases} \mathit{Std} \cup \mathit{Spec} \cup \mathit{BRef} \cup \mathit{FRef} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

where

$$\begin{aligned} \mathit{Std} &= \{i \mid \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_i)\} \\ \mathit{Spec} &= \{i \mid \texttt{/?} \in \mathsf{tags}(\omega_i)\} \\ \mathit{BRef} &= \{i \mid (\texttt{/<} \in \mathsf{tags}(\omega_i)) \wedge (\exists j \text{ s.t. } j < i \wedge \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_j))\} \\ \mathit{FRef} &= \{i \mid (\texttt{/>} \in \mathsf{tags}(\omega_i)) \wedge (\exists j \text{ s.t. } j > i \wedge \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_j))\} \end{aligned}$$

In the same spirit, the $\mathsf{subst}()$ function must be refined to transparently substitute referred terms for matched references. Given a term sequence $r$, a model $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, a statement $s = \langle \omega_1, \ldots, \omega_m \rangle$, and a valid match $m = \langle t_1, \ldots, t_n \rangle$ between $\mu$ and $s$, we define

$$\mathsf{subst}(\mu, m, r) = r \left[ {}^{\tau_1}\!/\mathsf{ref}(\tau_1, s, m, t_1) \right] \cdots \left[ {}^{\tau_n}\!/\mathsf{ref}(\tau_n, s, m, t_n) \right]$$

where

$$\mathsf{ref}(\tau, s, m, l) = \begin{cases} \omega_{\mathsf{bref}_<(\tau,s,m,l)} & \text{if } \texttt{/<} \in \mathsf{tags}(\omega_l) \\ \omega_{\mathsf{bref}_>(\tau,s,m,l)} & \text{if } \texttt{/>} \in \mathsf{tags}(\omega_l) \\ \omega_l & \text{otherwise} \end{cases}$$

and the best referred term among possibly many is determined by the following functions:

$$\begin{aligned} \mathsf{bref}_<(\tau, s, m, l) &= \begin{cases} \max\{i \in \mathit{Std} \mid i < \min m\} & \text{if } \exists i \in \mathit{Std} \text{ s.t. } i < \min m \\ \min\{i \in \mathit{Std} \mid i < l\} & \text{otherwise} \end{cases} \\ \mathsf{bref}_>(\tau, s, m, l) &= \begin{cases} \min\{i \in \mathit{Std} \mid i > \max m\} & \text{if } \exists i \in \mathit{Std} \text{ s.t. } i > \max m \\ \max\{i \in \mathit{Std} \mid i > l\} & \text{otherwise} \end{cases} \end{aligned}$$

For example, consider a statement $s$ and a rule $\rho$ defined by

$$\begin{aligned} s &= \langle \texttt{A/T}_1\texttt{/T}_2, \texttt{B}, \texttt{C/<} \rangle \\ \rho &= (\langle x/T_1, B, y/T_2 \rangle, \langle S, x, y \rangle, \langle U, y \rangle) \end{aligned}$$

The statement term `C/<` could stand either for $\mathtt{A/T_1/T_2}$ or for `B`. In fact, the matching map

$$\mathsf{match}(\rho, s) = \langle\{1,3\}, \{2\}, \{1,3\}\rangle$$

is complete, and produces the two valid matches $m_1 = \langle 1,2,3\rangle$ and $m_2 = \langle 3,2,1\rangle$. Of these, the cost model selects the first as the best one. The action to execute is computed as follows:

$$\begin{aligned}
&\mathsf{subst}(\mu, m, \langle S, x, y\rangle) = \\
&= \langle S, x, y\rangle \left[{}^{x}\!\middle/\mathsf{ref}(x/T_1, s, 1)\right] \left[{}^{B}\!\middle/\mathsf{ref}(B, s, 2)\right] \left[{}^{y}\!\middle/\mathsf{ref}(y/T_2, s, 3)\right] \\
&= \langle S, x, y\rangle \left[{}^{x}\!/\omega_1\right] \left[{}^{B}\!/\omega_2\right] \left[{}^{y}\!\middle/\omega_{\mathsf{bref}_<(y/T_2,s,3)}\right] \\
&= \langle S, x, y\rangle \left[{}^{x}\!/\omega_1\right] \left[{}^{B}\!/\omega_2\right] \left[{}^{y}\!/\omega_1\right] \\
&= \langle S, x, y\rangle \left[{}^{x}\!\middle/A/T_1/T_2\right] \left[{}^{B}\!/B\right] \left[{}^{y}\!\middle/A/T_1/T_2\right] \\
&= \langle S,\ A/T_1/T_2,\ A/T_1/T_2\rangle
\end{aligned}$$

Similarly, the computation of the substitution yields

$$\mathsf{subst}(\mu, m, \langle U, y\rangle) \quad = \quad \langle U,\ A/T_1/T_2\rangle$$

### 5.4.3 Wildcard tagging

Wildcard tagging, denoted by a `/*` tag, is a way to specify that a term in the model part of a rule should match any term in the statement that has at least a tag. This is different from simply using in the model a variable term with no tag: in that case, *any* term would be matched, by definition of $\mathsf{target}()$, including terms with an empty tag set. Formally, the definition of $\mathsf{target}()$ is extended once more:

$$\mathsf{target}(\tau, s) = \begin{cases} \mathit{Std} \cup \mathit{Spec} \cup \mathit{BRef} \cup \mathit{FRef} \cup \mathit{Wild} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

where *Std*, *Spec*, *BRef*, and *FRef* are the same as before, and *Wild* is defined as follows:

$$\mathit{Wild} = \begin{cases} \{i \mid \#\mathsf{tags}(\omega_i) \geq 1\} & \text{if } \texttt{/*} \in \mathsf{tags}(\tau) \\ \{\} & \text{otherwise} \end{cases}$$

To reflect the intuition that precise matches are more relevant than wildcard matches, we also update the definition of $\mathsf{mscore}()$:

$$\begin{aligned}
\mathsf{rscore}(\rho) \quad = \quad & ms * K_{TERMS} \\
& + ts * K_{TAGS} \\
& + cs * K_{CONSTS} \\
& + K_{RULE} \\
& + ws * K_{WILDS}
\end{aligned}$$

where $ws = \#\{i \mid \texttt{/*} \in \mathsf{tags}(\tau_i)\}$ and $K_{WILDS}$ is a parameter of the cost model.

As an example of application of wildcard tagging, consider a statement $s = \langle \texttt{A/T}_1, \texttt{B/T}_2, \texttt{C/<} \rangle$ and a rule $\rho = \langle x/T_1, y/* \rangle$. The matching map would be

$$\mathsf{match}(\rho, s) = \langle \{1, 3\}, \{1, 2, 3\} \rangle$$

with valid matches $m_1 = \langle 1, 2 \rangle$, $m_2 = \langle 1, 3 \rangle$, $m_3 = \langle 3, 1 \rangle$, $m_4 = \langle 3, 2 \rangle$ (of which $m_1$ has the best score).

## 5.5 Implementation issues and optimizations

In implementing the algorithms described in the previous section, particular care has been taken with respect to efficiency. In fact, a number of optimization techniques and heuristics have been introduced in the parsing algorithm.

Among the code-level optimizations, the following are particularly relevant:

- tokenization is applied to all input text. Each string is represented by an integer hash code (a token); tagged terms are represented by a token for the base term and by a sorted, variable-length, array of tokens for the tags. By using this representation, the function $\mathsf{target}(\tau, s)$ can be implemented as a comparison between two integers (if $\tau$ is a constant) or as a linear[4] comparison among (small) arrays of integers (if $\tau$ is a variable).

- the optimized code for the parser is less than 16Kb in size, so that it can reside entirely in most contemporary microprocessors' cache. Also, most data structures are aligned and arranged so that the access patterns of the parsing algorithm exhibits a high degree of locality, thus minimizing cache misses.

Such code-level optimizations, however, can only improve performance by a constant factor. More relevant improvements are obtained by algorithm-level optimizations, like the following:

- matches in which consecutive terms in the model point to terms in the statement that are more than MaxDistance terms apart each other are called *thin matches*, and are not considered valid (in the sense of Section 5.3.2). If a matching map does not contain at least a valid match that is not thin, the entire matching map is not considered complete (and thus the corresponding rule is not applicable).

- *early abandon* of a parse sequence that is being explored is activated when any of the following three conditions is met:

  1. the current score is worse than BadThreshold;

[4] Although better comparison strategies exists for ordered arrays, the typical size of a tag list is rather small — one to four elements —, making linear comparison faster in practice.

2. the current score is worse than LevelMult times the depth of the current parse sequence, plus BadThreshold;
3. the difference between the current score and the previous score (i.e., the score before the last parsing step) is worse than BadImprovement.

Each of the four heuristic optimizations cited above can be enabled or disabled independently, according to the needs of a particular configuration. In the standard configuration, thin matches refusal is disabled, while all three early abandon clauses are enabled.

## 5.6 Evaluation

### 5.6.1 Coverage

The particular applications at which Cico is directed make it difficult to formally define a measure for coverage. In classical usage, a parser for natural language is applied to a known corpus of text, and coverage is defined as the percentage of sentences from the corpus that have been parsed correctly. This measure is relevant under the assumption that the parser should parse correctly *any* sentence in the given language. This is a reasonable assumption for grammar-based parsing, but not for domain-based parsing. In fact, Cico is used in two contexts:

1. during requirements writing, with a fixed rule set. In this case there is no reference corpus, and instead the requirements writer is encouraged to rewrite unparsed or wrongly parsed requirements *that convey information in one of the known domains* in order to make them parseable. The assumption in this case is that Cico's rule sets constitute a style manual for the domain. Notice that Cico's inability to parse requirements that are not relevant for one of the known domain is not to be counted as a parsing failure. Rather, it should be considered either as an indication that the set of models in use is not sufficient with respect to the domain being addressed (e.g., there is no user interface model defined while writing usability requirements for a system), or that the unparsed requirement is not significant for the analysis (e.g., a requirement prescribes voltage constraints on certain output lines, whereas the requirements engineer is more interested in studying the system's time behaviour). Chapter 8 presents a case study on this kind of application.

2. during requirements analysis, with a fixed requirements document. In this case, there is no reference rule set, and instead a rule set is defined to cover the language used in a document, *thus defining a new domain* that Cico should be able to parse. Chapter 7 gives an account of a case study in which this application was relevant.

While no exact measure can be given of coverage, there is empirical evidence that CICO's coverage is adequate for the kind of documents at which it is aimed. Indeed, several dozens projects, both industrial and academic, have succeeded in obtaining complete parsing of requirements document describing a wide array of systems.

### 5.6.2 Performance

CICO's basic parsing algorithm does not guarantee convergence. In fact, any rule whose model matches its substitution, if fired once, would cause the algorithm to run forever. The obvious extension to cases where multiple rules are involved in a loop also applies.

However, sufficient conditions for convergence can be given. In particular, the basic parsing algorithm terminates on any input if the following condition holds on the rule set $R$:

$$\forall(\mu,\alpha,\sigma)\in R, \mathsf{len}(\mu) > \mathsf{len}(\sigma)$$

where the function $\mathsf{len}()$ is defined as follows:

$$\mathsf{len}(\langle a_1,\ldots,a_n\rangle) = n$$

This is indeed the case for the applications CICO is geared to. When building parse trees, the substitution is simply a 1-term reference to the parse tree for the substituted fragment (see Chapter 6), so that convergence is usually reached in only a few iterations (corresponding to a number of nested recursive calls of BasicParse() in the order of 10). Actually, heuristics, contexts and a restricted rule set make CICO's parse algorithm run almost in linear time in most cases.

To study CICO's performance in real cases, we analyzed a set of 2276 requirements (statements) from 32 projects, representative of CICO's normal usage, with a rule set consisting of 146 rules in various domains. The source text for those statements contained 26467 words, that were reduced to 18037 terms by input canonization and glossary substitution (that included stopwords removal). After canonization, statements ranged in length from 2 terms to a rather exceptional 62 terms. The histogram in Figure 5.1 shows how statements were distributed according to their length. As can be seen, most statements were only 3 to 7 terms long, corresponding to simple, atomic statements (e.g., "The server receives transactions from a terminal"). The second data series in Figure 5.1 refers to the average parse time for statements of a given length. Its distinctive features can be explained by considering changes that occur in input data when statements gets too long:

- the first peak, at statement length 17, corresponds to the introduction of context-delimiting structures in the text. The effect of the introduction of progressively more pervasive contexts can be observed up to length 25, when the increase in length once again offsets the gain obtained by the use of contexts;

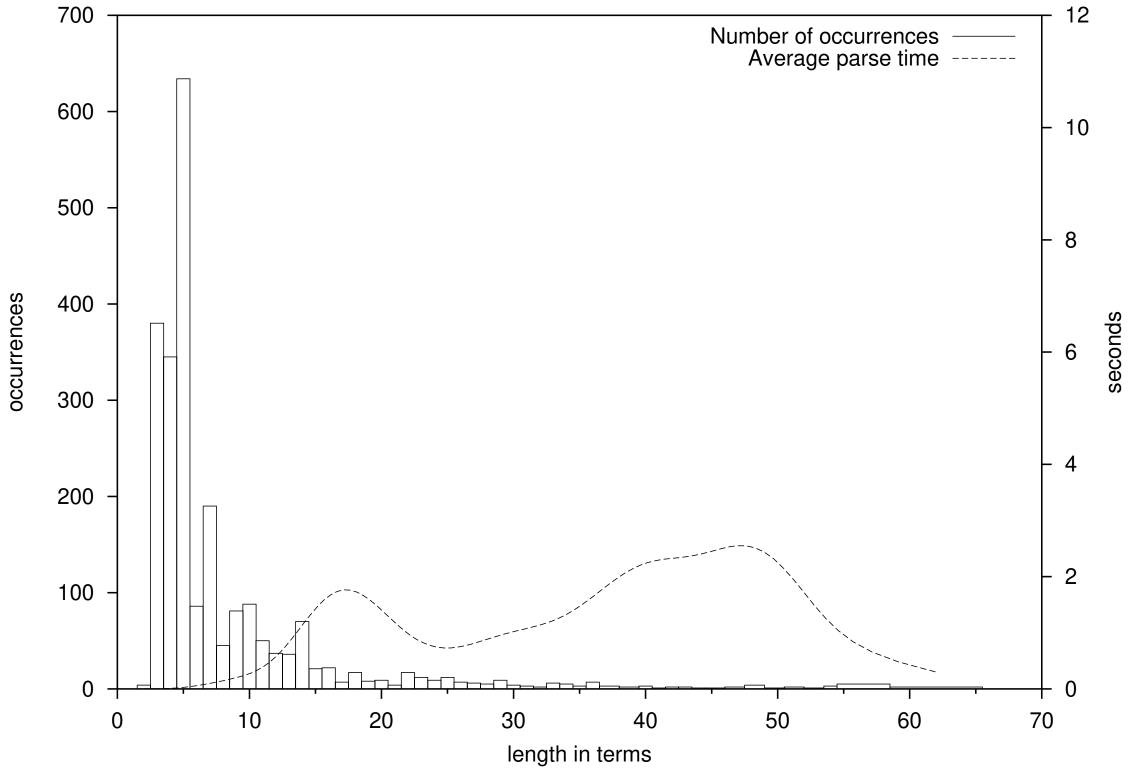



Figure 5.1: Frequency and parse times at different sentence length.

- the second peak, at statement length 47, corresponds to a sort of self-selective effects. In fact, as statements increase in length, the person writing them tends to split them in smaller pieces, with only the simplest (and thus, faster to parse) long statements remaining intact.

In both cases, experimental evidence indicate that a psychological threshold exists around 2-3 seconds of parse time, prompting changes in the statements when parse time for a statement grows above that limit.

A few observations can be made on the empirical data in Figure 5.1 to understand the scale of performance in a real-life situation:

- 80% of all requirements were 10 terms long or shorter; average parse time for those requirements was 0.048 seconds;

- 90% of all requirements were 15 terms long or shorter; average parse time for those requirements was 0.061 seconds;

- the longest requirement in our data set was 62 terms long. Average parse time for *all* requirements was 0.17 seconds

- average parse time for an entire requirement *document* was 12.58 seconds;

- all times were taken on a "slow" machine (running on a Pentium 133MHz processor from Intel).

While less than 13 seconds to parse a requirement document can be considered acceptable in batch processing, this time is still excessive for interactive processing. One technique to overcome this limitation consists in caching the results of a parse run and only re-parse changed or added statements when a subsequent version for the same document is submitted for parsing. Preliminary results on this *differential parsing* approach, based on the analysis of almost 200 submissions of a single requirement document during its evolution, indicate typical parse times for a new version of approximatively 1.2 seconds, versus an average 40 seconds for a complete re-parse — a speedup factor of 25.1, and well below the acceptability limit for interactive use.

## 5.7 Cost model parameters

Table 5.2 lists the actual numeric values for the parameters used by Cico's scoring system. The specific values shown have proved effective in practical use with the Italian language, but most of the parameters are non-critical, and can be adjusted to the needs of a particular language or application. For example, "rule score" in English is a very different concept from "score rule", and the value for $K_{INVERSION}$ should be adjusted accordingly. Also, the various thresholds for optimizations may be adjusted (and made more or less liberal) depending on the performance of the host machine on which Cico is run.

| **Parameter** | **Value** | **Used in** |
|---|---|---|
| $K_{DROPPED}$ | 1.0 | mscore() |
| $K_{DIST}$ | 0.1 | |
| $K_{INVERSION}$ | 0.6 | |
| $K_{TERMS}$ | 0.6 | rscore() |
| $K_{TAGS}$ | 0.2 | |
| $K_{WILDS}$ | -0.1 | |
| $K_{CONSTS}$ | 0.3 | |
| $K_{RULE}$ | 1.0 | |
| $K_{REST}$ | 0.6 | lscore() |
| VeryBadScore | -100.0 | algorithms |
| MinimumAcceptableScore | -4.0 | |
| BadThreshold | -4.0 | optimizations |
| LevelMult | 0.3 | |
| BadImprovement | -2.0 | |

Table 5.2: Parameter values for the scoring system.

# Chapter 6

# Models and language

Abstract

In this chapter we present the models currently available in CIRCE. Related models are collected into families which share the same ontology. We also present some MAS rules and associated tags that are used by CICO to initially populate the semantic repository of tuples needed by those models.

For certain models, we also describe desirable consistency properties of the model that are checked by related validation modelers, as well as the measures collected on the model by related metrication modelers.

A number of examples clarify how models are used and represented in CIRCE.

## 6.1 Entity models

We collect under the label *entity models* all those models whose purpose is to show which entities are present in the problem domain, and to detail their properties (attributes) and relations. Entity models do not express behaviour or functions, but may model the nature of data on which behaviour depends and that are operated upon by functions.

### 6.1.1 Language

The three main concepts in the entity models ontology are the classical E-R ideas of *entity*, *relation* (with a *cardinality*) and *attribute*. For each of them, CIRCE defines a tag that can be assigned to glossary entries that designate corresponding terms. In particular, an entity is indicated by the `/ENTITY` tag (that, however, can be omitted, although in this case some MAS rule will have a lower priority), the name of a relation is indicated by the `/REL` tag, and an attribute by an `/ATTR` tag. Cardinality of a relation can be indicated by numeric quantities (with a `/NUM` tag) or by numeric-type data (with an `/INF` tag, see Section 6.2).

Table 6.1 shows the parsing rules[1] which produce tuples relevant for the entity family of models. We recall that upper case terms indicate constants, while lower case terms indicate variables (Section 5.3.1); `$` is the variable substitution symbol (variables in the action and substitution parts are replaced by their values, established by matching models with the text of the requirements), and `/*` denotes the wildcard tag (Section 5.4.3) that matches any tag (but not the empty tag set). The special variable `$ID` represents the unique tuple identifier for the tuple output by the action part of a rule, and is used to reference tuples in complex structures. Finally, system glossaries provide a number of pre-defined terms for each category (e.g.: "is a part of" is a known `/REL`ation) and of synonyms for both variable and constant parts (e.g.: "each" and "every"). Word ordering is largely unconstrained, so the language recognized is often much more liberal than a simple look at the rules would seem to imply.

Relations are described by er*nm* tuples (where *nm* denotes cardinality) generated by statements matching the first and second group of rules shown in Table 6.1. While the first group of rules seems straightforward, it is actually lacking in two respects: first, it does not allow for enumerations (e.g., "the system controls three production cells and a supervisory unit"), and second, it is too general (due to the use of wildcard tagging) and can cause mismatches. The second group of rules, with the help of the two rules for conjunctive entity lists, suffers none of these defects, but needs an explicit `/ENTITY` declaration for terms whose cardinality is not defined. Since wildcard matches get a lower score under our scoring system (per Table 5.2), implementing both set of rules allows the requirements engineer to use either simple requirements with simple or no tagging, or complex requirements at the cost of some added `/ENTITY` tag.

We are especially concerned with extracting entity models even in the presence of imperfect or incomplete tagging because this kind of requirements ("world description" statements) are usually among the first to be stated by a specifier.

Tuples of type erxx and cardentity constitute a practical encoding for complex descriptive requirements, but their usage in modelers would be complicated by the need to follow several levels of indirection to get to the actual entity name and to the cardinality of the relation. For this reason, the *MiscER* modeler transforms erxx, cardentity and conj tuples related to entities into simpler `er11`, `er1n` and `ern1` tuples, acting as a demultiplexer.[2] Any other modeler in the system can thus only consider

[1]While the parser CICO is completely language-independent, the rule set currently used by CIRCE covers the Italian language. The rules we show are approximate translations of the Italian ones, and some linguistic subtlety may be lost in the process.

[2]To allow tracing back of the tuples it produces, *MiscER* also outputs erxxref tuples that link each "new" er*nm* tuple to the original erxx tuple it derives from. Many other modelers output similar "service" tuples; we will generally not mention them unless they are significant in the context.

| *Model* | *Action* | *Substitution* |
|---|---|---|
| **Basic relationships** | | |
| ent1/* relation/REL ent2/* | ER11 $ent1 $ent2 $relation | |
| ent1/* relation/REL card/NUM ent2/* | ER1N $ent1 $ent2 $relation $card | |
| EVERY ent1/* relation/REL ent2/* | ER11 $ent1 $ent2 $relation | |
| EVERY ent1/* relation/REL card/NUM ent2/* | ER1N $ent1 $ent2 $relation $card | |
| card/NUM ent1/* relation/REL ent2/* | ERN1 $ent1 $ent2 $relation $card | |
| **Complex relationships** | | |
| card/NUM ent/* | CARDENTITY $card $ent | $ID/ENTITY |
| ent1/* relation/REL ent2/ENTITY | ERXX $ent1 $ent2 $relation | |
| ent1/ENTITY relation/REL ent2/* | ERXX $ent1 $ent2 $relation | |
| ent1/ENTITY relation/REL ent2/ENTITY | ERXX $ent1 $ent2 $relation | |
| EVERY ent1/* relation/REL ent2/ENTITY | ERXX $ent1 $ent2 $relation | |
| EVERY ent1/ENTITY relation/REL ent2/* | ERXX $ent1 $ent2 $relation | |
| EVERY ent1/ENTITY relation/REL ent2/ENTITY | ERXX $ent1 $ent2 $relation | |
| **Attribute declaration and access** | | |
| ent/* HAS ATTRIBUTE attrib/ATTR | ATTR $attrib $ent | |
| EVERY ent/* HAS ATTRIBUTE attrib/ATTR | ATTR $attrib $ent | |
| attrib/ATTR IS ATTRIBUTE OF ent/* | ATTR $attrib $ent | |
| attrib/ATTR OF ent/* | ATTR $attrib $ent | $ID/INF |
| **Conjunctive /ENTITY and /ATTR lists** | | |
| ent1/ENTITY AND ent2/ENTITY | CONJ $ent1 $ent2 | $ID/ENTITY/CONJ |
| ent1/ENTITY ent2/ENTITY/CONJ | CONJ $ent1 $ent2 | $ID/ENTITY/CONJ |
| attr1/ATTR AND attr2/ATTR | CONJ $attr1 $attr2 | $ID/ATTR/CONJ |
| attr1/ATTR attr2/ATTR/CONJ | CONJ $attr1 $attr2 | $ID/ATTR/CONJ |
| **Navigation and selection** | | |
| ALL ent1/ENTITY OF ent2/ENTITY | SELECTALL $ent1 $ent2 * | $ent1 |
| ent1/ENTITY OF ent2/ENTITY | SELECT1 $ent1 $ent2 * | $ent1 |
| ALL ent1/ENTITY WHICH relation/REL ent2/ENTITY | SELECTALL $ent1 $ent2 $relation | $ent1 |
| ent1/ENTITY WHICH relation/REL ent2/ENTITY | SELECT1 $ent1 $ent2 $relation | $ent1 |

Table 6.1: Some MAS rules for entity models.

those simpler tuples in building its model.

Attribute declaration is handled by the first three rules in the attribute section of Table 6.1. The declaration of an attribute simply inserts an `attr` tuple into the repository. However, even simply mentioning an attribute of an object in any context (e.g., "...the system sends the *temperature of the cell* to the supervisory unit") acts as an implicit declaration, so that explicitly listing all the attributes of an object is usually redundant (but may be useful for validation purposes). Differently from explicit declarations, access to attributes of an object results in a `/INF` term to be substituted, meaning that the expression (e.g., "temperature of the cell") refers to something with an information content (as described in the next section).

Navigation among entities and selection of entities is expressed through the last group of rules. These allow for references like "...the system turns off *all the boilers of the selected cell* in at most 5 seconds", where a specific set of boilers is meant to be operated upon. The rules cover both generic cases ("boilers *of* a cell") and more precise statements ("boilers *that are part of* a cell"). In any case, the substitution replaces the selective expression with the selected object, i.e., the "boilers that are part of a cell" are still boilers.

We complete this section with an example encompassing most of the rules we introduced. Consider the statements

> The system controls three production cells and a supervisory unit.
> The system monitors the temperature of the production cell that is selected by the supervisory unit.

with a glossary containing

> system/SYS[3]: control system.
> temperature/ATTR.
> cell/ENTITY: production cell, production cells.
> supervisor/ENTITY: supervisory unit.
> selection/REL: is selected by.

(both "monitoring" and "control" are predefined as `/REL`ations in system glossaries). The parsing process for these statements is described in Table 6.2. The resulting tuple space, on which the modelers of the entity family can operate, is shown graphically in Figure 6.1.

### 6.1.2 Basic entity models

The `er`*nm* and `attr` tuples on their own already constitute useful models, and indeed the *Entity-Relationships* and *Attributes* projectors provide respectively a graphical representation of a traditional E-R model and an ordered list of known attributes for each object.

[3] `/SYS` tags specifies which terms denote the "system" part of our domain, i.e., what we are asked to design.

| | |
|---|---|
| The system controls three production cells and a supervisory unit. | *original text* |
| system/SYS control/REL 3/NUM cell/ENTITY and supervisor/ENTITY | *canonical form* |
| system/SYS control/REL **3/NUM cell/ENTITY** and supervisor/ENTITY | ⟨ 1,cardentity,3/NUM,cell/ENTITY ⟩ |
| system/SYS control/REL **@1/ENTITY and supervisor/ENTITY** | ⟨ 2,conj,@1/ENTITY,supervisor/ENTITY ⟩ |
| **system/SYS control/REL @2/ENTITY/CONJ** | ⟨ 3,erxx,system/SYS,@2/ENTITY/CONJ,control/REL ⟩ |
| The system monitors the temperature of the production cell that is selected by the supervisory unit. | *original text* |
| system/SYS monitor/REL temperature/ATTR of cell/ENTITY which selection/REL supervisor/ENTITY | *canonical form* |
| system/SYS monitor/REL temperature/ATTR of **cell/ENTITY which selection/REL supervisor/ENTITY** | ⟨ 4,select1,cell/ENTITY,supervisor/ENTITY,selection/REL ⟩ |
| system/SYS monitor/REL **temperature/ATTR of cell/ENTITY** | ⟨ 5,attr,temperature/ATTR,cell/ENTITY ⟩ |
| **system/SYS monitor/REL @5/INF** | ⟨ 6,er11,system/SYS,@5/INF,monitor/REL ⟩ |

Table 6.2: Parsing sequence for sample statements in Entity models

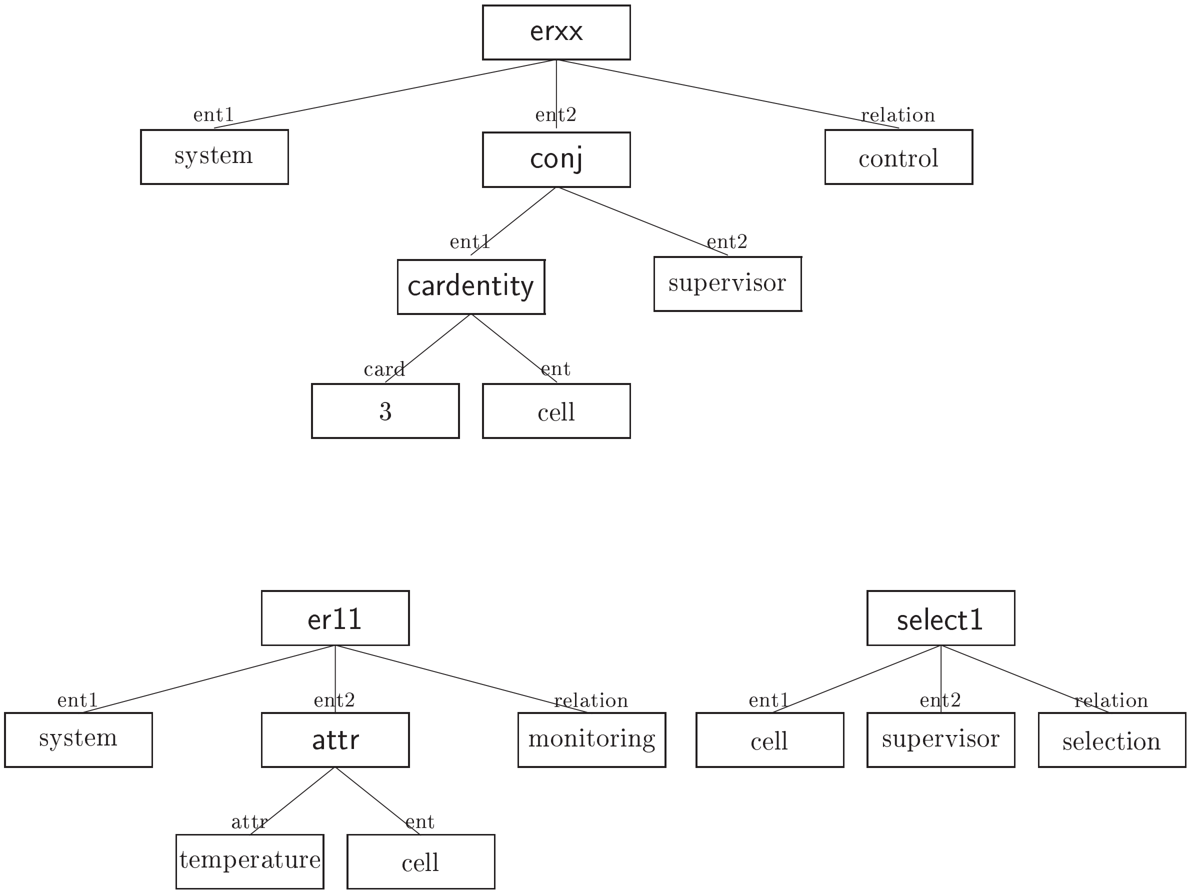



Figure 6.1: A forest of parse trees for the sample requirements (linkage, unparsing and service tuples omitted).

### 6.1.3 Classification and Composition

Two relationships are particularly relevant in modeling a domain, and thus deserve a special treatment. The first is the IS-A relationship, describing a classification hierarchy; the second is the PART-OF relationship, describing a composition hierarchy.

The corresponding modelers simply add is-a and part-of tuples for each er$nm$ tuple with the right relationship present in their input tuple space. Their behaviour can thus be described in logical style as

$$\mathsf{is\text{-}a}(X,Y) \leftarrow \mathsf{er}nm(X,Y,\text{IS-}\textsc{Aname})$$

for the *Classification* modeler, and

$$\mathsf{part\text{-}of}(X,Y) \leftarrow \mathsf{er}nm(X,Y,\text{PART-OF}\textsc{name})$$

for the *Composition* modeler. Both the is-a and the part-of model can be observed graphically through associate projectors as a forest of classification and composition trees. It is worthwhile to notice that these models do not assume an *interpretation* for the relationships they represent (i.e., inheritance properties or sharing of components); this is left to upper-level modelers.

The *BadIS-A* modeler is one of the components that assigns semantics to the is-a model. It assumes that inheritance holds between entities in a IS-A relationship, so properties of the ancestor should hold for the descendant. Since in Circe properties of entities are described by assigning tags to the terms that designate them, inheritance amounts to ensuring that a descendant has at least all the tags assigned to all its ancestors.

Formally, again in the logical style used above,

$$\mathsf{badis\text{-}a}(X,Y) \leftarrow \mathsf{is\text{-}a}(X,Y), \mathsf{tags}(Y) \not\subseteq \mathsf{tags}(X)$$

The badis-a model thus produced is used, among others, for validation purposes.

### 6.1.4 Subsumption

Given the er$nm$, attr, and is-a models, it is possible to identify cases where *subsumption* could be applied to simplify the requirements. Subsumption allows one to lift a common property of all instances of a class into the class itself, thus making it a general principle. In our context, the *Subsumption* modeler identifies opportunities for subsumption of relationships in a tuple space as follows:

$$\begin{aligned}
\mathsf{subsumerel}(X,Y,Rel,1,1) &\leftarrow \forall Z, \mathsf{is\text{-}a}(X,Z) \Rightarrow \mathsf{er11}(Z,Y,Rel) \\
\mathsf{subsumerel}(X,Y,Rel,1,N) &\leftarrow \forall Z, \mathsf{is\text{-}a}(X,Z) \Rightarrow \mathsf{er1N}(Z,Y,Rel,N) \\
\mathsf{subsumerel}(X,Y,Rel,N,1) &\leftarrow \forall Z, \mathsf{is\text{-}a}(X,Z) \Rightarrow \mathsf{erN1}(Z,Y,Rel,N) \\
\mathsf{subsumerel}(X,Y,Rel,N,M) &\leftarrow \forall Z, \mathsf{is\text{-}a}(X,Z) \Rightarrow \mathsf{erNM}(Z,Y,Rel,N,M)
\end{aligned}$$

Similarly, opportunities for subsumption of attributes are identified by

$$\mathsf{subsumeattr}(X, A) \leftarrow \forall Z, \mathsf{is\text{-}a}(X, Z) \Rightarrow \mathsf{attr}(A, Z)$$

While subsumption is a useful technique on its own that can help in writing more compact requirements, the subsumerel and subsumeattr models can also be regarded as hints to verify that the understanding of the problem by the requirement writer is correct and complete, by computing *approximate subsumption.*

In approximate subsumption, the condition that *all* the descendants of a given ancestor share a common relationship or attribute is relaxed. Instead, relationships and attributes that are shared by a significant number[4] of descendants are marked in the approxsubsumerel and approxsubsumeattr models for presentation to the user. Often, approximate subsumption is an indication that either the description of some entity is incomplete (e.g., it lacks an attribute it should have), or that the classification hierarchy is inconsistent (i.e., a distinction should be made between descendants that share some property and descendants that do not, by introducing intermediate designations). Typical cases in which approximate subsumption hints at defects in the requirements are shown in Figure 6.2.

In that figure, an attribute $a$ shared by all instances of Room is lifted up to the general class, and an approximate subsumption for attribute $b$, shared by some but not all instances, is found. The approximate subsumption could indicate incompleteness in requirements for the Student Room, that could be perfected by adding the missing attribute $b$ (which could then be subsumed). On the other hand, it could indicate an oversimplification in the class hierarchy, and the requirement writer could decide that a more detailed description (i.e., introducing collective names for "Rooms with $b$" and "Rooms without $b$") would be more appropriate. Obviously, the requirement writer could also decide that the description is acceptable as it is, for example because he or she knows that more room types will be added in the future. The purpose of lightweight validation checks like those performed by the *Subsumption* modeler is always to attract attention to possible problems and alternatives, not to dictate a strict design style to a competent professional.

### 6.1.5 Selection

The use of selective expression like "all boilers of a cell" can be checked for existence and uniqueness against other entity models. This is the purpose of the *BadSelect* modeler.

As Table 6.1 shows, it is possible to express selection either through a specific relationship (e.g., "all boilers that are part of a cell") or through anonymous reference (e.g, "all boilers of a cell"). In the first case, the mentioned relationship must

[4]This is defined as a user-controlled percentage of the total number of descendants.

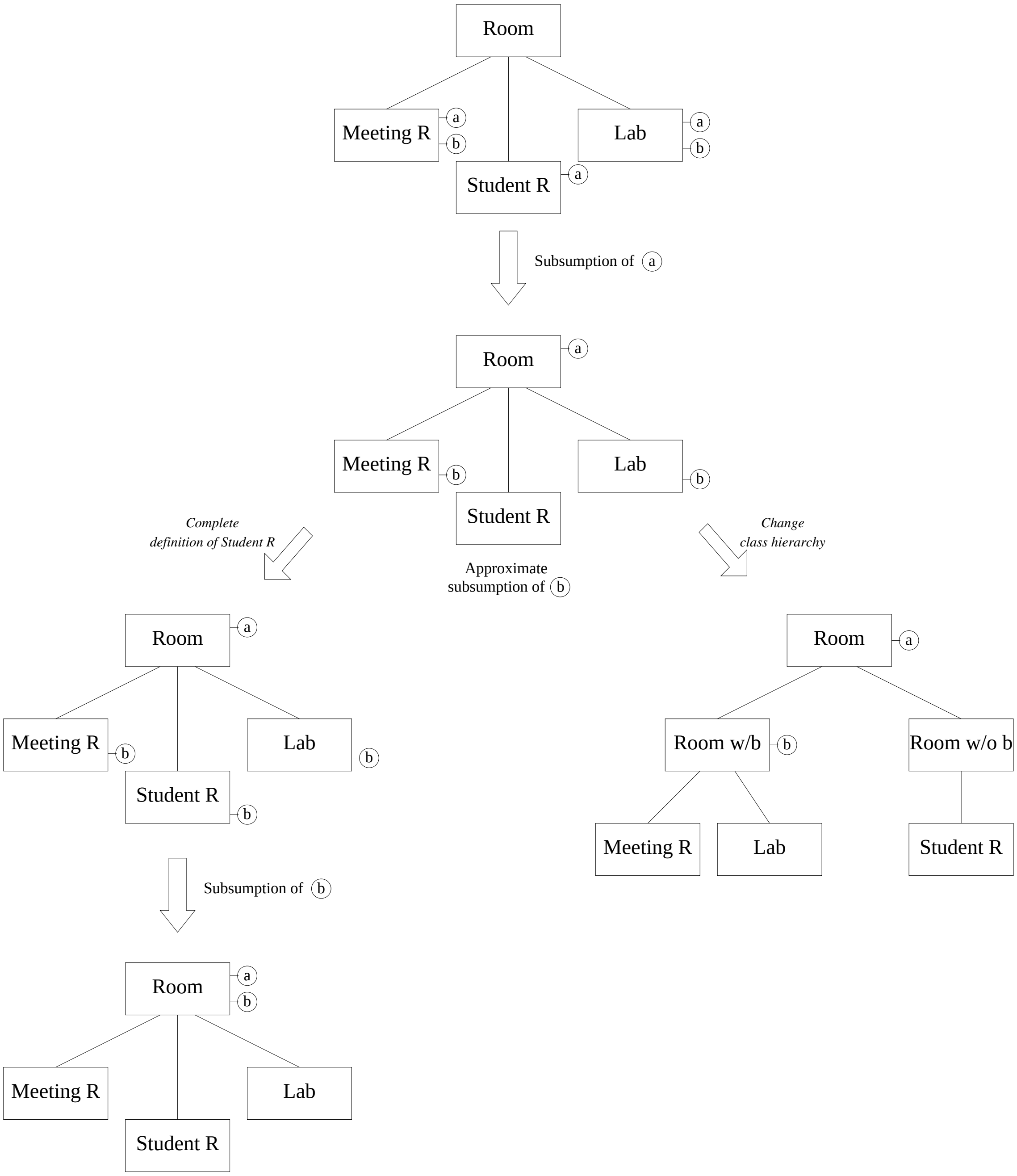



Figure 6.2: Effect of subsumption and approximate subsumption on a class hierarchy.

exist:[5]

$$\begin{aligned}\mathsf{badselectnorel}(X,Y,R) \;\leftarrow\;& R \neq *, \quad \mathsf{select1}(X,Y,R) \vee \mathsf{selectall}(X,Y,R),\\ & \nexists \mathsf{er11}(X,Y,R), \quad \nexists \mathsf{er1n}(X,Y,R,N),\\ & \nexists \mathsf{ern1}(X,Y,R,N), \quad \nexists \mathsf{ernm}(X,Y,R,N,M)\end{aligned}$$

In the second case, a suitable relation must exist, i.e., a relation that can be navigated from the given entity to the selected entity. As an example, given that "a cell contains three boilers" is known, "the cell of a given boiler" is well-identified, as is the selection "all the boilers of a given cell", while "the boiler of a given cell" is not a valid selection. Moreover, the suitable relation must be unique, to avoid ambiguity in the interpretation. Consider what happens when only the following two relations are known:

- a cell contains three boilers;
- a cell operates one boiler (at a time).

The selective expression "the cell of the boiler" is now ambiguous, and could be taken to mean either "the cell that contains the boiler" or "the cell that operates the boiler".[6] On the other hand, the expression "the boiler of the cell" can be interpreted only as "the boiler operated by the cell", since "the boiler contained in the cell" would be ambiguous. For the same reason, "all the boilers of the cell" could be interpreted either as "all the boilers contained in the cell", or as "all the boilers operated by the cell", or even as "both the boilers contained and those operated by the cell" — a degree of ambiguity that deserve flagging as a potential problem.

To identify such cases, the *BadSelect* modeler first builds a *reduced E-R graph* by only considering navigable relations:

$$\begin{aligned}\mathrm{rer}(X,Y,R) &\leftarrow \mathsf{er11}(X,Y,R)\\ \mathrm{rer}(X,Y,R) &\leftarrow \mathsf{erN1}(X,Y,R)\\ \mathrm{rer}(X,Y,R^{-1}) &\leftarrow \mathsf{er1N}(X,Y,R)\end{aligned}$$

(notice that rer() is only an internal representation of the modeler and does not constitute a model, and that the $R^{-1}$ notation identifies the *name* of the inverse relation of $R$). Then, the models badselectnorel (no suitable relationship exists) and badselectmulrel (too many suitable relationships exist) are populated as follows:

$$\begin{aligned}\mathsf{badselectnorel}(X,Y) &\leftarrow \mathsf{select1}(X,Y,*), \nexists R : \mathrm{rer}(X,Y,R)\\ \mathsf{badselectmulrel}(X,Y) &\leftarrow \mathsf{select1}(X,Y,*), \exists R,R' : \mathrm{rer}(X,Y,R), \mathrm{rer}(X,Y,R'), R \neq R'\end{aligned}$$

[5]Here and in the following we will not specifically mention that the is-a model is taken into account when searching for specific tuples. A support library provides transparent (but optional) inheritance handling to modelers and projectors.

[6]Note that the boiler referred to could well be the same in both cases, but we have no way of assuring this without a thorough understanding of the environment.

Inconsistencies in the selectall tuples are identified in a similar way.

The *BadSelect* modeler presented above only handles *direct* references. However, in discourse we often take shortcuts, i.e., assuming that each job uses a single boiler, instead of

> ... the cell containing the boiler that is used by the current job ...

we could say, as long as no ambiguity can arise,

> ... the cell used by the current job ...

or even

> ... the cell of the current job ...

Correct interpretation of such selection expression require a more in-depth analysis of our er$nm$ model. This is the purpose of the *BestSelect* modeler. When presented with a selection for which no direct relationship can be found, the modeler performs a shortest path search on the reduced E-R graph to identify possible interpretations. Shortest paths correspond to less indirect references, i.e., in our previous example "the cell containing the boiler that is used by the current job" is an interpretation that is more plausible than "the cell that contained the boiler that was used by the last job scheduled by the operator that is scheduling the current job".

The corresponding model is bestselect$(X, R_1, Z_1, \ldots, R_n, Y)$ where each $R_i, Z_i$ represent a relation and an intermediate object in the path to the intended selected entity. As above, the absence of any navigable path from the given entity to the selected entity constitutes an inconsistency, while the presence of multiple paths with the same length constitutes an ambiguity.

In presenting the views derived from the models discussed in this section, the absence of a suitable direct or indirect resolution for the selection expression is considered a serious error, as are ambiguities either in direct or indirect resolution. Indirect resolutions are always presented to the requirement writer as "reminders" for verification and documentation purposes.

## 6.2 Functional models

*Functional models* are concerned with the computational aspects of a system. The main focus for such models is to capture the computation performed by the system from an input/output perspective; the system itself is seen as a function transforming input values into output values or "results". Functional models ignore temporal or dynamic aspects of the system; no control flow is described nor considered.

In most cases, requirements that feed functional models are written in the early stages of the requirements analysis phase, immediately following domain description statements. Functional models are also useful in interaction with the customer, since they make it easy to check that the correct interfaces have been defined for the system.

### 6.2.1 Language

In a functional model, *data* can be *received* or *sent*, and can be used in *computations* resulting in new data. In CIRCE, terms describing data are given the `/INF` tag (meaning "the term has information content"). Entities working on this data can be marked `/IN` (if they are capable of receiving data), `/OUT` (if they are capable of sending data) and `/ELAB` (if they are capable of performing computations on the data).

In addition, a particular kind of asynchronous data exchange is introduced by the tags `/IRQ` (meaning "can request interrupts") and `/RIRQ` (meaning "accepts interrupt requests").

Relational, unary and binary[7] operators are assigned the `/RELOP`, `/UNOP` and `/BINOP` tags, respectively. These tags can be used to describe conditions, as in "is greater than/RELOP", "is out of bound/UNOP", or "is in/BINOP". They will be used mainly in behaviour models, but are introduced here since the access to data items that they imply is relevant for the validity of data flow models.

Two more tags express properties that are not used by MAS rules, but that are useful for a number of modelers. These are the `/SYS` tag that we already met, and that is used to mark parts of the system that the requirements writer is going to specify, and the `/BLTIN` tag that is used to characterize fixed data that is known to a (sub)system (e.g., a table of cosines or the binary value representing a command). A subtle but important distinction exists between assigning a `/SYS` tag to the term designating a component and declaring in a requirement that the component "is part of" our main system. In the former case, we state that we are interested in modeling the internals of the component, while in the latter we merely state that the component is in administrative, architectural or implementation sense a piece of our system. In the absence of a `/SYS` tag, we are not interested in its internals, and will treat it as a black box.

This distinction is relevant for our functional models because it induces two different borders for the system along which we can describe interaction. The first border separates our system from its environment; the second border separates the parts we are concerned with from what we are given as "already done".

The language that can be used to described functional aspects of a system is varied, and so are the rules contained in CIRCE's rule set. Table 6.3 shows a few of them, representative of the various categories. Requirement writers can actually use a much larger set of alternative expressions, producing the same tuples as the ones shown. The `/EVT` and `/ACT` tags used in the substitution parts stand for "this is an event" and "this is an action", respectively, and will be discussed in more depth in Section 6.3.1.

[7] Operators with arity greater than two are seldom found in ordinary language, and if needed they can be treated as computations, with an unary operator applied to the result. This has a beneficial side effect in that it is made clear in the requirements who, when and how the operator is applied.

| *Model* | *Action* | *Substitution* |
|---|---|---|
| **Basic input and output** | | |
| target/IN RECEIVES data/INF FROM sender/OUT | RECV $target $sender $data | $ID/EVT |
| sender/OUT SENDS data/INF TO target/IN | SEND $sender $target $data | $ID/EVT/ACT |
| sender/OUT DISPLAY data/INF ON device/IN | DISPLAY $sender $device $data | $ID/EVT/ACT |
| sender/OUT SETS data/INF FOR receiver/IN | SETFOR $sender $receiver $data | $ID/EVT/ACT |
| client/IN/OUT REQUESTS data/INF FROM server/IN/OUT | DQUERY $client $server $data | $ID/EVT/ACT |
| server/IN/OUT SENDS data/INF UPON REQUEST FROM client/IN/OUT | DQUERY $client $server $data | $ID/EVT/ACT |
| **Interrupts** | | |
| petitioner/RIRQ INTERRUPTS someone/IRQ | IRQ $petitioner $someone | $ID/EVT/ACT |
| someone/RIRQ IS INTERRUPTED BY petitioner/IRQ | IRQ $petitioner $someone | $ID/EVT |
| **Computations** | | |
| agent/ELAB COMPUTES result/INF USING params/INF | ELAB $agent $params $result | $ID/EVT/ACT |
| agent/ELAB COMBINES params/INF/CONJ OBTAINING result/INF | ELAB $agent $params $result | $ID/EVT/ACT |
| agent/ELAB MAXIMIZES value/INF | OPTCRIT MAX $agent $value | |
| agent/ELAB MINIMIZES value/INF | OPTCRIT MIN $agent $value | |
| agent/ELAB OPTIMIZES value/INF | OPTCRIT OPT $agent $value | |
| agent/ELAB INSERTS item/INF INTO collection/INF | ELABINSERT $agent $item $collection | $ID/EVT/ACT |
| agent/ELAB REMOVES item/INF FROM collection/INF | ELABREMOVE $agent $item $collection | $ID/EVT/ACT |
| agent/ELAB MODIFIES data/INF USING params/INF | ELABMODIFY $agent $data $params | $ID/EVT/ACT |
| agent/ELAB COPIES source/INF INTO destination/INF | ELABCOPY $agent $source $destination | $ID/EVT/ACT |
| **Operators** | | |
| val1/INF op/RELOP val2/INF | RELOP $val1 $op $val2 | $ID/EVT |
| val/INF op/UNOP | UNOP $val1 $op | $ID/EVT |
| val1/INF op/BINOP val2/INF | BINOP $val1 $op $val2 | $ID/EVT |
| Conjunctive rules for /INF, /IN, /OUT omitted | | |

Table 6.3: Some MAS rules for functional models.

### 6.2.2 Data flow model

The first step in obtaining a data flow model from the tuples produced by our parser consists in identifying the flow of data implicit in the different, detailed tuples that are in the semantic repository.

This is the purpose of the *DataFlow* and *Elabs* modelers, that abstracts from the many details of communications and computations that are recorded by the rules in Table 6.3 to produce a simpler model described by only two tuple types:

$$\mathsf{dflow}(sender, receiver, data)$$

and

$$\mathsf{elab}(agent, parameters, result)$$

This abstraction is accomplished by dropping unneeded information, as in

$$\begin{array}{rcl}
\mathsf{dflow}(S,R,D) & \leftarrow & \mathsf{send}(S,R,D)\\
\mathsf{dflow}(S,R,D) & \leftarrow & \mathsf{recv}(R,S,D)\\
\mathsf{dflow}(S,R,D) & \leftarrow & \mathsf{display}(S,R,D)\\
\mathsf{dflow}(S,R,D) & \leftarrow & \mathsf{setfor}(S,R,D)\\
\mathsf{dflow}(S,R,\text{“request for” } D) & \leftarrow & \mathsf{dquery}(S,R,D)\\
\mathsf{dflow}(S,R,D \text{ “upon request”}) & \leftarrow & \mathsf{dquery}(R,S,D)\\
\mathsf{dflow}(S,R,\text{“irq”}) & \leftarrow & \mathsf{irq}(S,R)\\
\mathsf{elab}(A,P.R,R) & \leftarrow & \mathsf{elabinsert}(A,P,R)\\
\mathsf{elab}(A,P.R,R) & \leftarrow & \mathsf{elabremove}(A,P,R)\\
\mathsf{elab}(A,P.R,R) & \leftarrow & \mathsf{elabmodify}(A,R,P)\\
\mathsf{elab}(A,P,R) & \leftarrow & \mathsf{elabcopy}(A,P,R)
\end{array}$$

The $P.R$ notation makes a single conjunctive list out of items in $P$ and $R$, which can themselves be conjunctive lists. As for the is-a model, a library function provides transparent handling of such lists to the modelers.

The projectors for the data flow model need only access the dflow and elab rules directly produced by MAS rules or synthesized by the modelers above, and also the optcrit tuples, that need special treatment. In particular, the “irq”, “request for $D$”, and “$D$ upon request” labels given above represent special markers that allow the projectors to use a different graphical style for those flows.

Circe provides several different views on the data flow model represented by dflow and elab tuples. In the *High-level data flows* view, all system components are collapsed into a single component, thus hiding intra-system flows and computations and highlighting instead interactions of the system with its environment and its net computations on input data. The *Low-level data flows* view, on the other hand, shows also intra-system flows, thus exhibiting more details on the architecture of the system.[8]

By default, data is not considered shared, i.e., each component of the system that accesses some data has a private copy of the data item, and synchronization among the copies, if and when desired, must be explicitly described in the requirements. This is the most common case for control systems, in which system components often correspond to physically distinct components. Data can also be declared `/SHARED`, in which case it is assumed that different (logical) components can access the same instance of that data — with locking and synchronization issues demanded to some unspecified low-level support. This is indeed the case when different components are allocated on the same physical node, and thus have access to a common shared memory.

As an example, consider the transaction processing system described by the following statements:[9]

1. When the front-end receives a query from a client,
   - if the query is in the cache, the front-end sends the query to the cache manager,
   - otherwise, the front-end sends the query to the dbms.
2. The cache manager retrieves the result from the cache using the query.
3. The cache manager returns the result to the client that sent the query.

where *query*, *result* and *cache* are defined as `/INF`, while *client*, *front-end*, *cache manager* and *dbms* are declared `/IN/OUT/ELAB` (with *front-end* and *cache manager* also being system components and having the `/SYS` tag).

The *High-level data flow* and *Low-level data flow* views of the system described above are shown in Figure 6.3 and 6.4, respectively. As can be observed, in the low-level view two different instances of *query* exists, one in the front-end space, and the other in the cache manager space. Had *query* been declared `/SHARED`, the views would have shown a single instance of the *query* data store, contained in the intersection between the *Front-end* and *Cache manager* boxes.

The various data flow models are built by Algorithm 5, which is a simplified version of the algorithm used by the actual data flow projectors. In that algorithm, given a data item $d$ (that could be an `/INF` item produced by parsing rules, e.g.,

[8] The architecture we refer to here can be the physical architecture of the system, and in this case a detailed requirements document will dictate constraints on the design phase (as is often the case for physically distributed system), but may also be the logical architecture of the system, in which the structure of the system is used as a way to organize the description of functionalities and will not be mapped onto the design- or implementation-level architecture.

[9] This example is an excerpt from a longer specification; it uses linguistic constructs that will be discussed in the next sections. We concentrate here only on the parts that describe the flow of data.

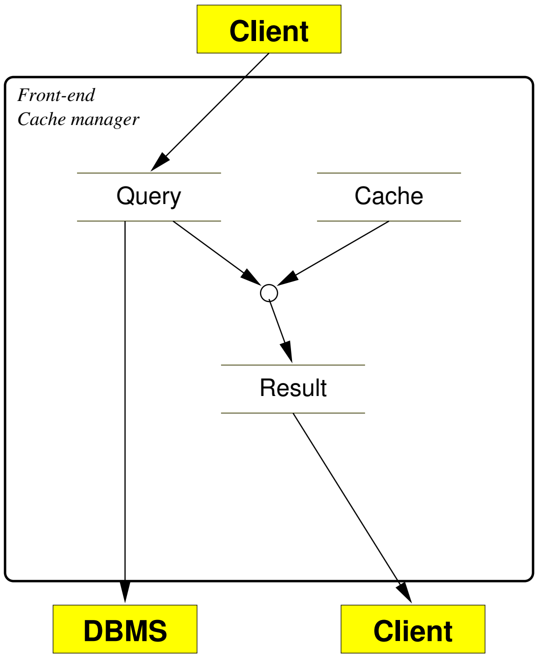



Figure 6.3: A sample *High level data flow* view.

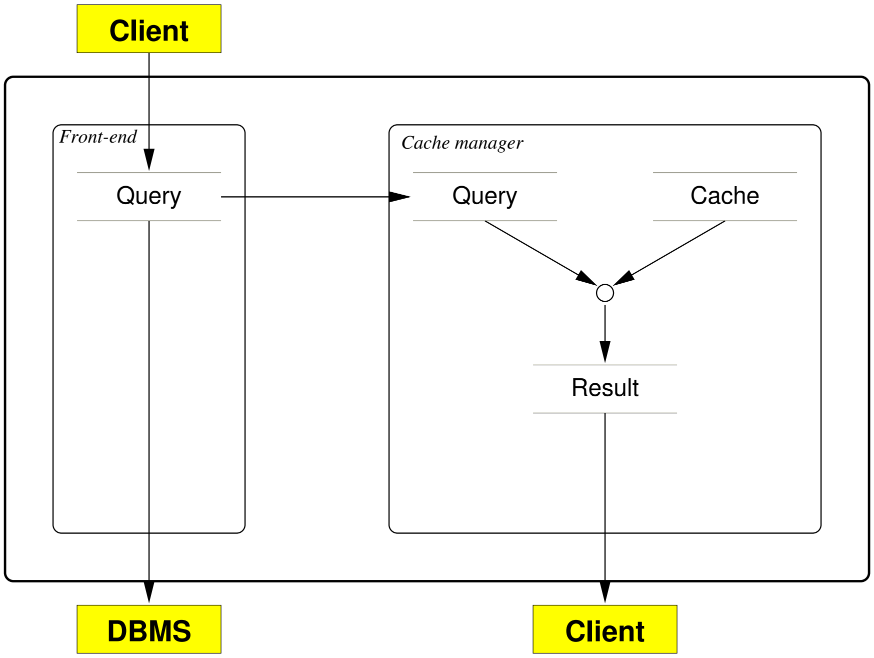



Figure 6.4: A sample *Low level data flow* view.

an attribute of some entity), the notation $\widehat{d}$ represents its *basic datastore*, i.e., the most elementary information item in the structure of $d$. In our case, $\widehat{d}$ is defined as follows:

$$\widehat{d} = \begin{cases} d & \text{if } d \text{ is a simple } \texttt{/INF} \text{ term} \\ \widehat{o} & \text{if } d \text{ is a reference to } \mathsf{attr}(a, o) \text{ and } o \text{ is an } \texttt{/INF} \\ \mathsf{unparse}(d) & \text{otherwise} \end{cases}$$

where unparse() is a library function that produces a textual representation of a parse (sub)tree, by using service tuples output by Cico during the parsing process.

When the basic datastore is different from the data item, a label is placed on the corresponding edge to identify the attribute of the item that is flowing across the edge. Also, labels are placed on function nodes (the small circles representing computations) identifying the agent of the corresponding elab tuple. The interaction environment provides controls for selectively showing or hiding these labels, to reduce display clutter.

In Algorithm 5 the $c\!:\!n$ notation is used to denote a node $n$ constrained inside a component $c$, distinct from any other node with the same label in other components[10], as the *Query* datastore in Figure 6.4.

The output of Algorithm 5 is a graph $G = (E, N)$, where the edges $E$ (which can have specific drawing styles and labels) are as specified by the "add arc" commands, while the nodes $N$ are either input nodes, output nodes, datastores or function nodes (with each type having a distinctive graphic style and placement and a label). Datastores and function nodes also have an optional container; a partial order exists between containers according to the part-of model of Section 6.1.3, and is visually represented by graphical containment of container boxes.

Data flow models offer several opportunities for validation; a few interesting properties are listed below:

- *Liveness*, i.e., every data item mentioned in the description of the system is used in some way. This means that either it is output outside of the component where it is stored, or it is used in a computation or to determine behaviour. Formally, liveness is equivalent to requiring that for all $c : d$ in a data flow model obtained from a semantic repository $\mathcal{T}$, at least one of the following conditions hold:

    1. there exists a path from $c\!:\!d$ to some output node $o$
    2. $\mathsf{optcrit}(X, c, d) \in \mathcal{T}$
    3. $\mathsf{relop}(X, O, d) \in \mathcal{T} \quad \vee \quad \mathsf{relop}(d, O, Y) \in \mathcal{T}$
    4. $\mathsf{binop}(X, O, d) \in \mathcal{T} \quad \vee \quad \mathsf{binop}(d, O, Y) \in \mathcal{T}$

[10] For /SHARED /INF nodes $c$ can be a set of components instead of a single component; we omit related details in Algorithm 5 for the sake of clearness.

**Algorithm 5** Generic *Data flow* projector.

Given a semantic repository $\mathcal{T}$:

```
for all dflow(F, T, D) ∈ 𝒯 do
   for all f ∈ F, t ∈ T, d ∈ D do
      if issys(f) then
         if ¬issys(t) then
            add arc from datastore d̂ to output node t
         else {only in a Low-level view}
            add arc from datastore f : d̂ to datastore t : d̂
         end if
      else if issys(f) then
         if ¬issys(t) then
            add arc from input node f to datastore d̂
         end if
      end if
   end for
end for

for all elab(A, P, R) ∈ 𝒯 do
   for all a ∈ A, r ∈ R do
      if issys(a) then
         let e be a unique anonymous function node labeled a {in a High-level view}
         let a : e be a unique anonymous function node {in a Low-level view}
         for all p ∈ P do
            add arc from datastore p̂ to e {in a High-level view}
            add arc from datastore a : p̂ to a : e {in a Low-level view}
         end for
         add arc from e to datastore r̂ {in a High-level view}
         add arc from a : e to datastore a : r̂ {in a Low-level view}
      end if
   end for
end for
```

where $\text{issys}(n) = (/\mathsf{SYS} \in \mathsf{tags}(n))$

5. $\mathsf{unop}(d, O) \in \mathcal{T}$

- *Determinateness*, i.e., every data item used by the system is either a constant or is input or computed in some way. Formally, for all $c : d$ in a data flow model obtained from a semantic repository $\mathcal{T}$, at least one of the following conditions must hold:

  1. there exists a path from some input node $i$ to $c{:}d$
  2. $\texttt{/BLTIN} \in \mathsf{tags}(d)$
  3. $\mathsf{optcrit}(X, c, d) \in \mathcal{T}$

- *Focus*, i.e., avoiding redundant information. In our case, this means that describing data flows and computations in which the parts of the system that we are specifying are not involved is a loss of focus. Formally, the following properties must hold:

  1. $\forall s, \texttt{/ELAB} \in \mathsf{tags}(s) \Rightarrow \texttt{/SYS} \in \mathsf{tags}(s)$
  2. $\forall \mathsf{dflow}(X, Y, D) \in \mathcal{T}, \texttt{/SYS} \in \ \mathsf{tags}(X) \cup \mathsf{tags}(Y)$

  (the second property is more easily checked in the commpath model described below).

- *True sharing*, i.e., data declared as `/SHARED` must be accessed by at least two different components. Formally,

$$
\begin{aligned}
&\forall d, \texttt{/SHARED} \in \mathsf{tags}(d) \Rightarrow \exists c, c' \text{ s.t.}\\
&\quad \texttt{/SYS} \in \mathsf{tags}(c) \wedge \texttt{/SYS} \in \mathsf{tags}(c') \wedge\\
&\quad (\mathsf{dflow}(c, X, d) \vee \mathsf{dflow}(X, c, d) \vee (\mathsf{elab}(c, R, P) \wedge d \in R \cup P)) \wedge\\
&\quad (\mathsf{dflow}(c', X, d) \vee \mathsf{dflow}(X, c', d) \vee (\mathsf{elab}(c', R, P) \wedge d \in R \cup P)) \wedge\\
&\quad c \neq c'
\end{aligned}
$$

In the properties above, notice that — due to the definition of $\widehat{d}$ — a datastore $d$ is considered written to either when it is directly used to store some input value or result of a computation, or when at least one of its attributes is written to. The dual consideration holds for reading from a datastore. This does not cover the case when only an attribute $a$ of a value is written to, and only a different attribute $b$ of the same value is read from. While in formal terms this situation appears undesirable and easily detectable, in practice it cannot be discovered without a substantial amount of domain knowledge on the nature of the attributes and of the dependencies among them. Consider for example the relation that ties the payload of a truck with its weight, or how changes in the income of an employee may change his or her social security status. Rather than forcing the requirement writer to specify in such detail the relations among attributes of a single entity, we chose more pragmatically to relax liveness and determinateness with respect to different attributes.

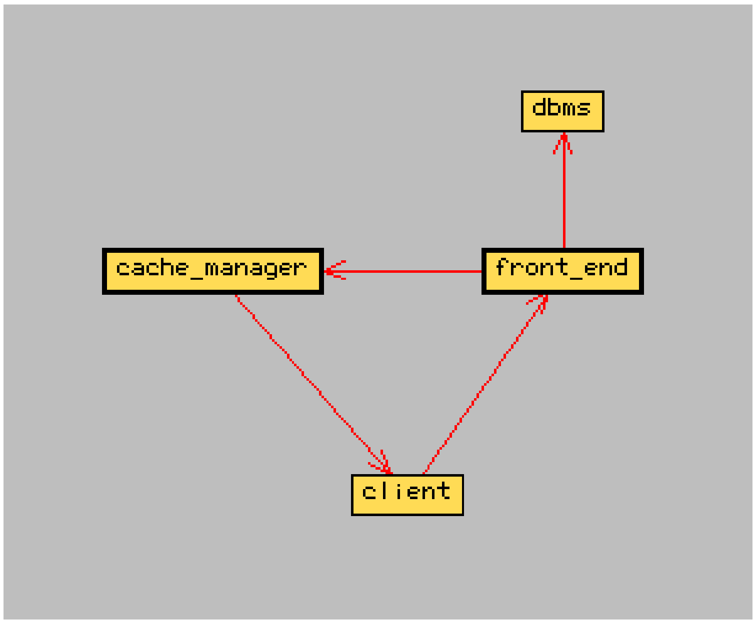



Figure 6.5: High-level communication paths view for the requirements on page 89.

### 6.2.3 Communication model

The *Communication paths* modeler synthesizes a more abstract version of the data flow models. The commpath model it computes is simply obtained by discarding internal details (e.g., elab and optcrit tuples) and data-related information from the dflow model, thus focusing on the *communication paths* in the system:

$$\mathsf{commpath}(X, Y) \leftarrow \mathsf{dflow}(X, Y, D)$$

For example, the commpath model for the system we described at page 89 is

commpath(client,front-end)
commpath(front-end,dbms)
commpath(front-end,cache manager)
commpath(cache manager,client)

From this model, the *Communications* projector produces a graph in which directed edges represent communications paths, while nodes represent components (with system components identified by a double strike border). Also in this case, two variants are available. The low-level view shows the part-of hierarchy by nesting boxes according to the composition model; the high-level view only shows first-level system components. An example of a high-level communications view can be seen in Figure 6.5, a screenshot from CIRCE.

The communication paths model is often particularly useful when trying to understand and correctly model the structure of the system, given its intuitiveness and the ease with which it can be discussed with the customer. Despite its simplicity, it is possible to state some interesting validation property on it.

For example, the part relative to communications of the *Focus* property given above can be re-stated in simpler terms as

$$\nexists \mathsf{commpath}(X, Y) \text{ s.t. } (\texttt{/SYS} \notin \mathsf{tags}(X) \wedge \texttt{/SYS} \notin \mathsf{tags}(Y))$$

i.e., our requirements do not describe data flows outside our scope of interest. Also, we want to ensure that our requirements describe a *single* system, not a set of unrelated parts. This can be expressed formally by asking that the intra-system communications submodel

$$\mathsf{commpath}(X, Y) \text{ s.t. } \texttt{/SYS} \in \mathsf{tags}(X) \wedge \texttt{/SYS} \in \mathsf{tags}(Y)$$

forms a connected graph when edges are undirected, or, in other terms, that there is a path between each pair of `/SYS` nodes when directionality of communications is not considered. The rationale for asking this property is that a system containing completely separated components is in fact just a collection of several detached systems, that should be described each in its own requirement document.

## 6.3 Behaviour and time models

The *behaviour* of a system is described by the causal and temporal relationships among the actions it performs and between external events and its actions. Behaviour models concentrate on describing, validating and measuring these aspects, abstracting from the nature of the events and actions that are introduced in the requirements.

### 6.3.1 Language

The main concepts in behaviour models are *events*, denoted by the tag `/EVT`, and *actions*, denoted by the tag `/ACT`. We have already seen in Table 6.3 how most data flow oriented expressions (e.g., sending data or completing a computation) constitute both an action (for the entity performing it) and an event (for entities observing it).[11] Entities that have an observable state are denoted by the `/HASSTATE` tag; entities that can actively change their state are assigned the `/ELAB` tag, since voluntarily changing one's own state requires a decision and thus computation capabilities (as a side effect, the focus property we introduced on page 93 ensures that requirements do not indulge in describing the inner reasons for state changes of non-`/SYS` objects). Terms describing a state (e.g., "on" and "off") are given the `/MODE` tag.

[11] Notice however that some expression, e.g. receiving data, is only considered an event and not an action, since it cannot be actively caused by the interested entity.

| *Model* | *Action* | *Substitution* |
|---|---|---|
| **Observable states** | | |
| obj/HASSTATE IS mode/MODE | TESTMODE $obj $mode | $ID/EVT |
| obj/HASSTATE IS IN mode/MODE | TESTMODE $obj $mode | $ID/EVT |
| obj/HASSTATE BECOMES mode/MODE | TESTMODE $obj $mode | $ID/EVT |
| obj/HASSTATE SWITCHES TO mode/MODE | TESTMODE $obj $mode | $ID/EVT |
| **Mutable states** | | |
| obj/HASSTATE/ELAB BECOMES mode/MODE | SETMODE $obj $mode | $ID/EVT/ACT |
| obj/HASSTATE/ELAB SWITCHES TO mode/MODE | SETMODE $obj $mode | $ID/EVT/ACT |
| **State declarations** | | |
| obj/HASSTATE CAN BE modeset/MODE/DISJ | MODESET $obj $modeset | |
| obj/HASSTATE CAN BE IN modeset/MODE/DISJ | MODESET $obj $modeset | |
| obj/HASSTATE IS INITIALLY mode/MODE | INITMODE $obj $mode | |
| obj/HASSTATE IS INITIALLY IN mode/MODE | INITMODE $obj $mode | |
| **Disjunctive rules** | | |
| mode1/MODE OR mode2/MODE | DISJ $mode1 $mode2 | $ID/MODE/DISJ |
| mode1/MODE mode2/MODE/DISJ | DISJ $mode1 $mode2 | $ID/MODE/DISJ |
| mode1/MODE OR IN mode2/MODE | DISJ $mode1 $mode2 | $ID/MODE/DISJ |
| mode1/MODE IN mode2/MODE/DISJ | DISJ $mode1 $mode2 | $ID/MODE/DISJ |

Table 6.4: Some MAS rules for objects with states.

As shown in Table 6.4, observing a state change constitutes an event, while voluntarily changing state is both an event (for those observing) and an action. Possible states of an object can be declared, and in this case some further checking is possible; otherwise it is assumed that an object can assume all the states mentioned in conjunction with the object in the requirements.

### 6.3.2 Time model

Events and actions do not exist in a void, but take place in *time*. CIRCE uses a modular-linear time model, whose main concepts are instants (`/PIT`, from "point-in-time"), time spans (`/TSPAN`) and intervals (`/TINTERV`). Table 6.5 shows a sample of the available rules that produce or make use of time-related terms (CIRCE's rule set covers a much wider array of expressions).

While CIRCE in itself does not provide an interpretation for the time expressions we use, associate modelers define a semantics for them. Time is modeled as a linear sequence of instants, or time spans of length 0. On this sequence, several measuring systems are used to determine the initial and final instants of time spans. Certain measuring units are commensurable (e.g., seconds and hours), while others are not (e.g., hours and months). Since even in technical language time expressions like "a month" can mean different things (e.g., four to five weeks, or approximately 30 days, or one twelfth of a solar year), our time model uses different granularities for different problems, as shown in Table 6.6. Instants and time spans are comparable only if they are in the same scale, thus it is generally required that a requirements document uses a single scale.

CIRCE determines the time scale to use based on the `tspan` tuples in the repository. If any tuple mentioning units of an hour or smaller is found, the small scale is selected; otherwise the number of references for each time unit is computed, and the scale that has as base unit the most referenced unit is selected. Using more than a single scale (e.g., measuring time both in seconds and in months in the same requirement document) is not considered an error, but a warning is generated, and events defined on different scales are not considered comparable.

A `/PIT` term in this model designate a single point on the time axis. `/TSPAN`s designate the length (but not the starting or ending point) of a segment on the time axis, while a `/TINTERV` term designates a segment (starting and ending at known points). Expressions like *every x*`/TSPAN` designate the infinite set of modular repetitions of the given time span $x$, according to the natural modulo for the given unit in the time scale in use.

Events can be instantaneous or span a certain amount of time. In the second case, when they are used to indicate time instants either their starting time is considered (as in "When something happens,...") or their ending time (as in "After something has happened,..."), depending on the context. In addition, events that are also actions are considered as happening when the action is complete (i.e., at their ending time). In cases when this interpretation is not adequate, rules like `END`

| *Model* | *Action* | *Substitution* |
|---|---|---|
| **Causal dependencies** | | |
| `WHEN event/EVT action/ACT` | `DEPT $event $action` | `$ID/EVT/ACT` |
| `IF event/EVT action/ACT` | `DEPC $event $action` | `$ID/EVT/ACT` |
| `IF event/EVT THEN action/ACT` | `DEPC $event $action` | `$ID/EVT/ACT` |
| **Timed dependencies** | | |
| `span/TSPAN AFTER event/EVT` | `TSPANAFTEREVT $span $event` | `$ID/PIT` |
| `span/TSPAN SINCE event/EVT` | `TSPANAFTEREVT $span $event` | `$ID/PIT` |
| `span/TSPAN BEFORE event/EVT` | `TSPANBEFOREEVT $span $event` | `$ID/PIT` |
| `event/EVT DURING interval/TINTERV` | `EVTWHILEINTERV $event $interval` | `$ID/EVT` |
| `event/EVT BEFORE instant/PIT` | `EVTBEFOREPIT $event $instant` | `$ID/EVT` |
| `event/EVT AFTER instant/PIT` | `EVTAFTERPIT $event $instant` | `$ID/EVT` |
| `event/EVT FOR span/TSPAN` | `EVTFORSPAN $event $span` | `$ID/EVT` |
| `action/ACT AT instant/PIT` | `ACTATPIT $action $instant` | `$ID/ACT/EVT` |
| `action/ACT instant/PIT` | `ACTATPIT $action $instant` | `$ID/ACT/EVT` |
| `action/ACT EVERY span/TSPAN` | `REPEATEDACT $action $span` | `$ID/ACT` |
| `action/ACT EVERY span/TSPAN SINCE instant/PIT` | `REPEATEDACT $action $span $instant` | `$ID/ACT` |
| `action/ACT WHILE event/EVT` | `WHILEACT $action $event` | `$ID/ACT` |
| `action/ACT UNTIL event/EVT` | `UNTILACT $action $event` | `$ID/ACT` |
| **Time measures** | | |
| `A SECOND` | `TSPAN 1/NUM SECS` | `$ID/TSPAN` |
| `n/NUM SECONDS` | `TSPAN $n SECS` | `$ID/TSPAN` |
| `n/INF SECONDS` | `TSPAN $n SECS` | `$ID/TSPAN` |
| `A MINUTE` | `TSPAN 1/NUM MINS` | `$ID/TSPAN` |
| `n/NUM MINUTES` | `TSPAN $n MINS` | `$ID/TSPAN` |
| `n/INF MINUTES` | `TSPAN $n MINS` | `$ID/TSPAN` |
| `AN HOUR` | `TSPAN 1/NUM HRS` | `$ID/TSPAN` |
| `n/NUM HOURS` | `TSPAN $n HRS` | `$ID/TSPAN` |
| `n/INF HOURS` | `TSPAN $n HRS` | `$ID/TSPAN` |
| (milliseconds, days, weeks, months, and years omitted) | | |
| **Conjunctive and disjunctive rules** | | |
| `a/ACT AND b/ACT` | `CONJ $a $b` | `$ID/ACT/CONJ` |
| `a/ACT b/ACT/CONJ` | `CONJ $a $b` | `$ID/ACT/CONJ` |
| `a/EVT AND b/EVT` | `CONJ $a $b` | `$ID/EVT/CONJ` |
| `a/EVT b/EVT/CONJ` | `CONJ $a $b` | `$ID/EVT/CONJ` |
| `a/EVT OR b/EVT` | `DISJ $a $b` | `$ID/EVT/DISJ` |
| `a/EVT b/EVT/DISJ` | `DISJ $a $b` | `$ID/EVT/DISJ` |
| (additive conjunction of `/TSPAN`s omitted) | | |

Table 6.5: Some MAS rules for behaviour models.

| Scale | Unit | Definition |
|---|---|---|
| Small | millisecond | 1/1000 seconds |
| | second | base unit |
| | minute | 60 seconds |
| | hour | 3600 seconds |
| | day | 86400 seconds |
| Medium | day | base unit |
| | week | 7 days |
| | month | 28–31 days |
| | year | 365–366 days |
| Large | day | 1/30 months |
| | week | 1/4.35 months |
| | month | base unit |
| | year | 12 months |

Table 6.6: Time systems used by CIRCE.

`OF event/EVT` and `START OF event/EVT` substitute a `/PIT` term corresponding to the start instant or to the end instant of the event. Figure 6.6 shows graphically the standard interpretation for the time-related tuples introduced in Table 6.5.

For example, consider the following requirement for an imaginary in-flight missile control system:

> Ten seconds after the system has received a launch confirmation command, it shall turn the main engine on until missile altitude reaches cruise altitude.

Other rules in CIRCE's repertoire will record that "the system has received a launch confirmation command" is an event (say $e_1$), as is "missile altitude reaches cruise altitude" (we will call this $e_2$), and that "the system shall turn the main engine on" is as action ($a_1$). Also, "ten seconds" will be recognized as a time span ($t_1$) and produce a $\mathsf{tspan}(10, \text{secs})$ tuple; the new structure of the statement is thus

$t_1$ after $e_1$, $a_1$ until $e_2$.

Now, "$a_1$ until $e_2$" is recognized as an action ($a_2$), and produces the tuple $\mathsf{untilact}(a_1, e_2)$, while "$t_1$ after $e_1$" is recognized as an instant, that we will call $p_1$, producing a $\mathsf{tspanafterevt}(t_1, e_1)$ tuple. Our requirement is thus reduced to

$p_1$, $a_2$

which triggers the rule for ACTATPIT producing an $\mathsf{actatpit}(a_2, p_1)$ tuple.

The parsing tree for our requirement, corresponding to the generated tuples, is shown in Figure 6.7. Its interpretation according to our model is the following operational-style specification:

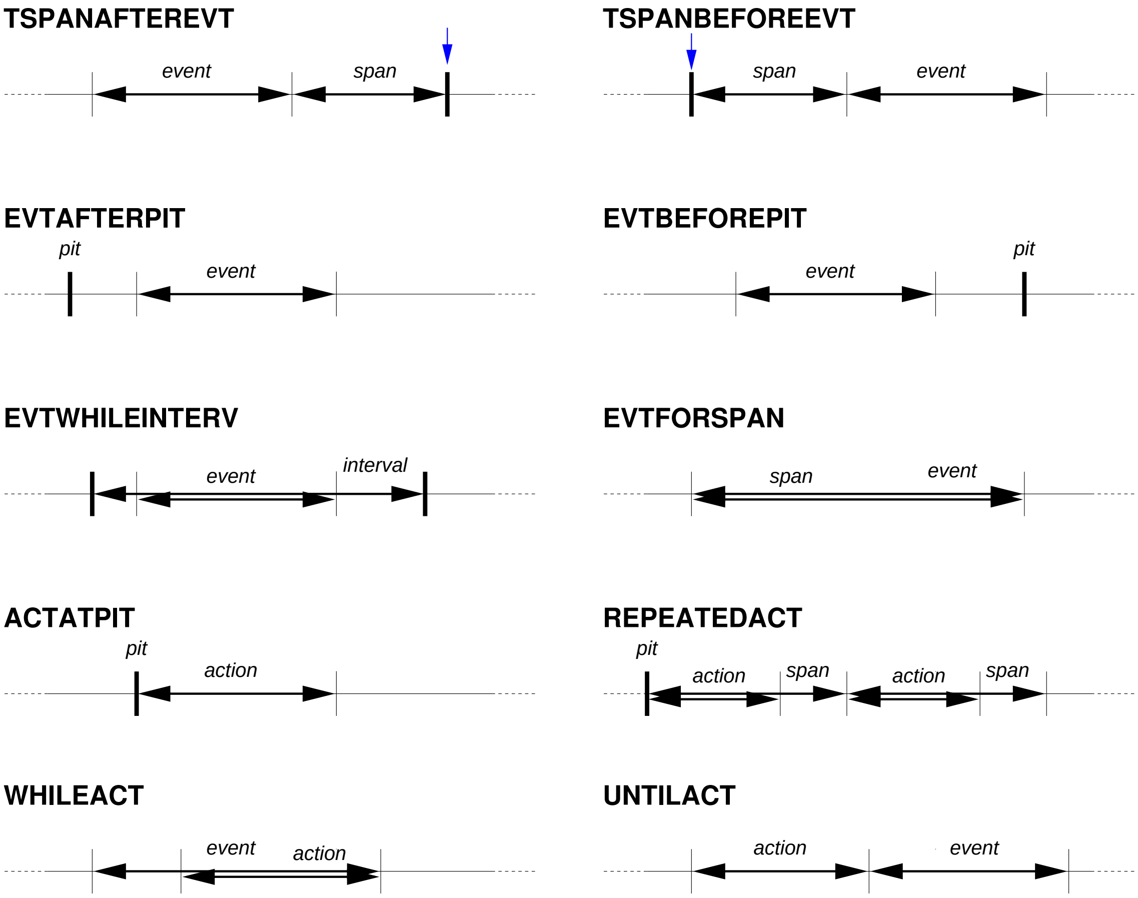



Figure 6.6: Interpretation of various tuples in the time model.

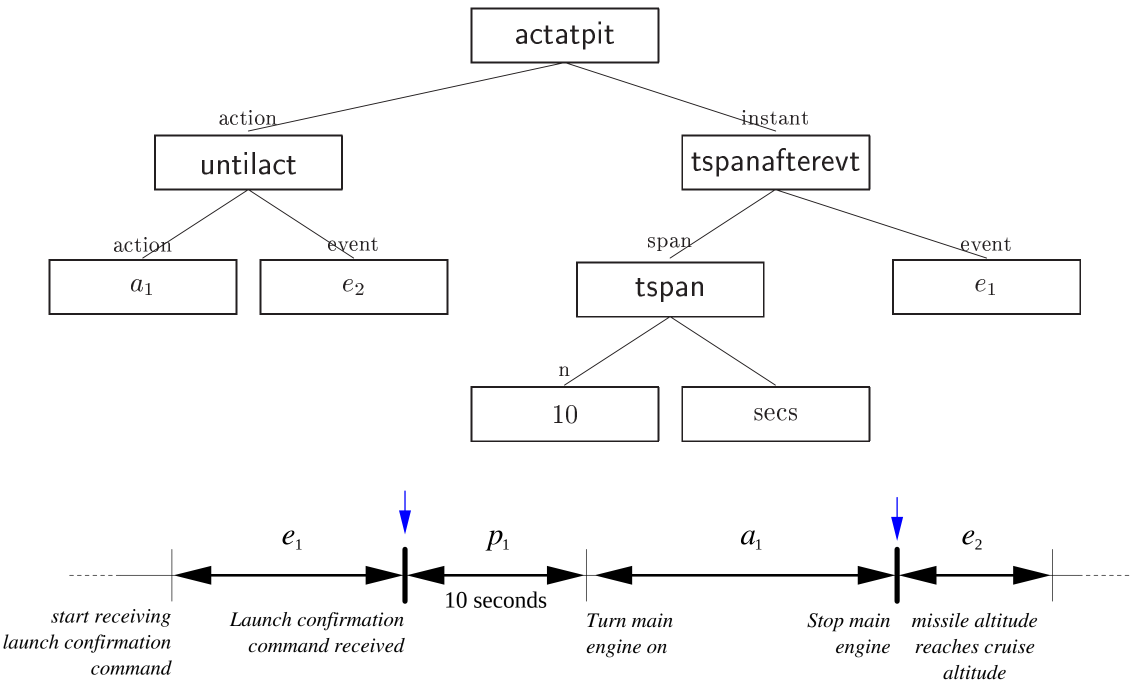



Figure 6.7: Parsing tree for a sample requirement and its interpretation.

1. wait until a launch confirmation command is received;
2. wait 10 seconds;
3. turn on main engine;
4. wait until missile altitude is greater than cruise altitude;
5. stop turning on main engine (i.e., turn off main engine).

The corresponding time diagram is at the bottom of Figure 6.7. It may be interesting to notice the different treatment of event-to-instant transformations in the cases of $e_1$ and $e_2$. In fact, $e_1$ is used in a tspanafterevt tuple, and as such the ending instant is taken as trigger. On the contrary, $e_2$ appears in a untilact tuple, and the starting instant is selected.

Time models are rendered in Circe through the *Sequences* projector. In the view it produces, each entity is represented by a timeline, with time flowing from the top to the bottom, on which events relevant for that entity are placed. Events are indicated by an arc connecting the action in the initiator timeline with the event in the observer timeline. In the case of communications, this corresponds to an arc from the sender's timeline to the receiver's timeline, labeled with the description of data exchanged. Delays are modeled by arcs connecting the initial and final event of the time span, labeled by their length.

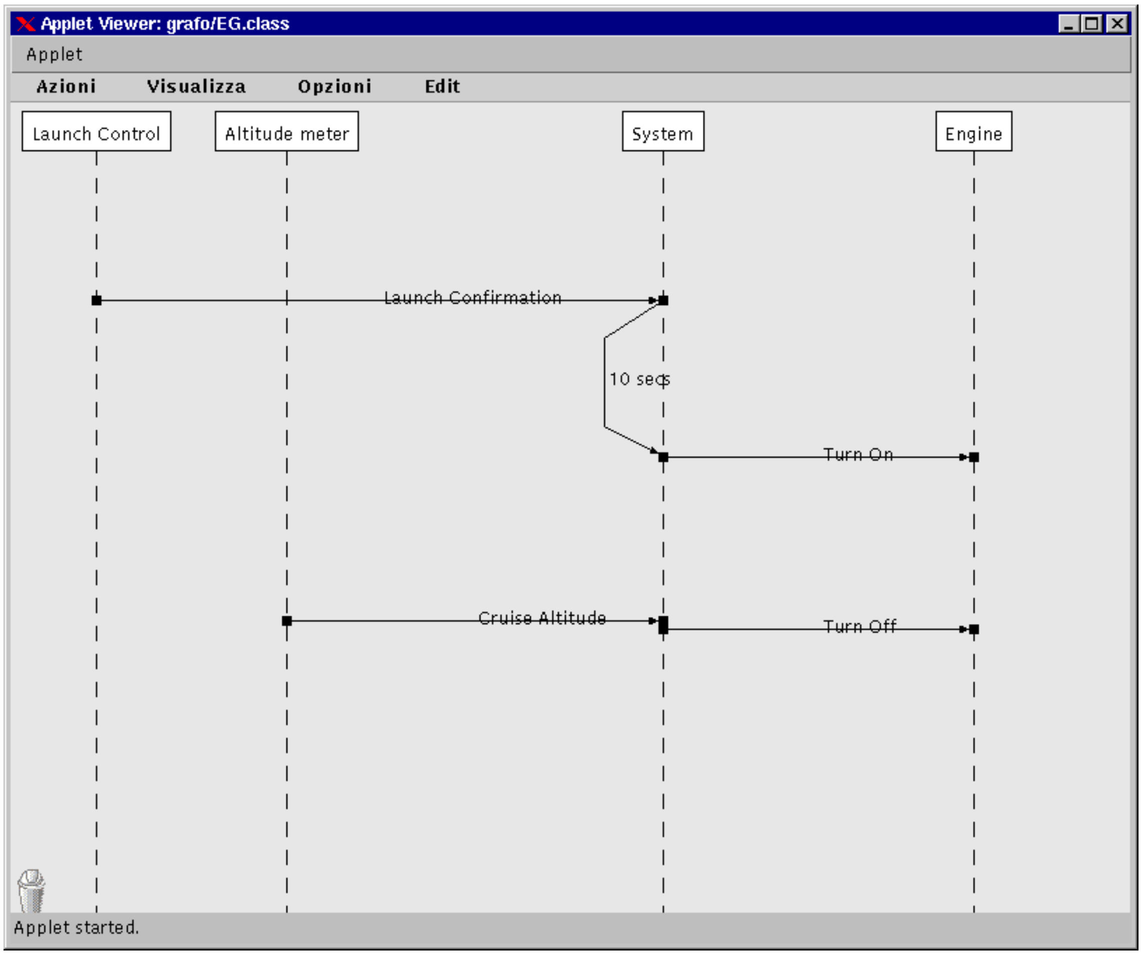



Figure 6.8: A sample *Time sequence* view.

A system of constraints is established on the produced diagram, so that the time (Y) coordinate of any observed event is the same of that of the corresponding action,[12] and that the final instant of a delay cannot precede its initial instant. These constraints are applied both to the automatic layout algorithm, and to user's manipulations of the interactive diagram.

Figure 6.8 shows the sequence view for a slightly more elaborate version of the requirement of page 99. Notice how the "System" timeline corresponds to the time diagram at the bottom of Figure 6.7. The system of constraints established on the diagram shown allows the user to rearrange freely the horizontal ordering of the timelines, but ensures that the starting and ending point of each arc connecting different timelines have at every moment the same Y coordinate. Moreover, the ending point of the 10 seconds delay is at a fixed vertical distance from the starting

[12]This constraint is applied under the assumption that events can be observed instantaneously. In cases where this assumption is not applicable, a constraint stating that the observation of an event is always successive to the event should be used instead.

point of the delay, where the distance corresponds to 10 seconds on the scale in use (here, the small scale of Table 6.6). The initial point of the "Turn On" event immediately follows the end of the delay, as the initial point of the "Turn Off" event follows the end of the "Cruise Altitude (reached)" event.

Except for these constraints, the user can thus rearrange the diagram (e.g., moving the start point of the "Cruise Altitude" event along the time axis). Notice however that our requirements do not state that the "Cruise Altitude" event must follow the "Turn On" event. As part of our common sense knowledge, we expect that a missile will not reach cruise altitude unless its engine is turned on, but this assumption is not made explicit in the requirements, and both the diagram and the user's ability to manipulate it reflect this fact.

### 6.3.3 Events and actions

Causal dependencies among events and actions can be expressed through statements like "if *event* then $action_1$ else $action_2$" or "when *event*, *action*".

When multiple conditions or nested dependencies are used, determining exactly under which conditions a certain action is taken may become difficult. For this purpose, the *Event-Action table* modeler builds a canonical representation of such dependencies in the xdep model:

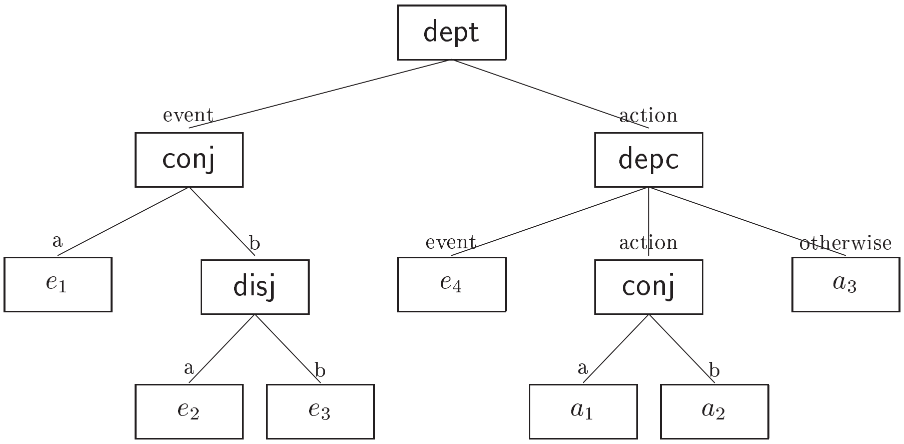
$$\mathsf{xdep}(E, A)$$

means that actions in the set $A$ depend on events and conditions in the set $E$. This is obtained by traversing parsing trees expressing dependencies, and collecting events and actions along the way. At each node, the *event* part can contain conjunctive and disjunctive lists of events, while the *action* part can contain conjunctive lists of actions (and possibly nested dependencies).

As an example, consider the following statement, where $e_i$ stands for an event and $a_i$ stands for an action:

When $e_1$ and either $e_2$ or $e_3$, if $e_4$ then $a_1$ and $a_2$, otherwise $a_3$.

The corresponding parse tree is shown in Figure 6.9. The statement is really expressing in a compact way a number of different cases, that the *Event-Action table* modeler describes in the four tuples at the bottom of Figure 6.9. A couple of related projectors uses the xdep model to unparse the events and actions and display them in tabular format to the user. In the first view, the table is ordered by conjunction of conditions, (Figure 6.10(a)), while in the second view it is ordered by action (Figure 6.10(b)). Both views help the requirement writer in checking its understanding of the consequences of different conditions (a) and of the preconditions for each action (b). Explicit tabular format is also particularly useful in interaction with the customer, since it does not require previous knowledge on his or her part.




$$\begin{aligned}
&\mathsf{xdep}(\{e_1, e_2, e_4\}, \{a_1, a_2\})\\
&\mathsf{xdep}(\{e_1, e_2, \neg e_4\}, \{a_3\})\\
&\mathsf{xdep}(\{e_1, e_3, e_4\}, \{a_1, a_2\})\\
&\mathsf{xdep}(\{e_1, e_3, \neg e_4\}, \{a_3\})
\end{aligned}$$

Figure 6.9: Canonic form for dependencies tuples.

| Events and conditions | Actions |
|---|---|
| $e_1$, $e_2$, $e_4$ | $a_1$, $a_2$ |
| $e_1$, $e_2$, $\neg e_4$ | $a_3$ |
| $e_1$, $e_3$, $e_4$ | $a_1$, $a_2$ |
| $e_1$, $e_3$, $\neg e_4$ | $a_3$ |

(a)

| Actions | Events and conditions |
|---|---|
| $a_1$ | $e_1$, $e_2$, $e_4$ |
| $a_1$ | $e_1$, $e_3$, $e_4$ |
| $a_2$ | $e_1$, $e_2$, $e_4$ |
| $a_2$ | $e_1$, $e_3$, $e_4$ |
| $a_3$ | $e_1$, $e_2$, $\neg e_4$ |
| $a_3$ | $e_1$, $e_3$, $\neg e_4$ |

(b)

Figure 6.10: Event-action tables corresponding to the parse tree in Figure 6.9.

Event-action tables can also be expressed as Event-Condition-Action tables similar in spirit to those used by the SCR [55] method. It is assumed that temporal dependencies (dept tuples) represent the occurrence of an event, while causal dependencies (depc tuples) represent a condition that must hold; an example of Event-Condition-Action table is given in Table 7.2 on page 140.

Event-action tables are well suited for consistency and completeness checks. However, some domain expertise is usually required to identify problems related to these models in a specification. Without domain expertise, it is generally not possible to judge on admissibility of specific event/action occurrences. Section 7.4.2 provides an illustration of what can be done regarding validation of models in the events-actions family when some specific knowledge on the nature of events and actions that occur in a certain domain is available.

### 6.3.4 Finite state automata

We have already introduced, while discussing the language used by behaviour models, the concepts of *state* and *entity with state*. Expressions in Table 6.4 allow the requirement writer to express state transitions of such objects; the *Automata* modeler uses the corresponding tuples to build a model of those transitions.

The finite state automata (FSA for short) model is based on the xdep model we described above. In detail, a state transition is added to the model whenever a dependency is found in xdep that has among its conditions the permanence of an entity in a certain state, and among its actions the transition to a different state; the remaining conditions are used to label the transition. It is also possible that no previous state is specified in a xdep tuple. In this case, it is assumed that the transition can take place, regardless of the previous state, whenever all the other conditions hold. If the entity that is being modeled has $n$ states, this is modeled as a set of $n-1$ possible transitions from all the other known states of the entity to the state mentioned in the action part. This strategy is described formally in Algorithm 6. The algorithm uses the $\cong$ "deep matching" operator. The operator returns true if a tuple matching its right argument is found in the tuple tree identified by its right argument. Formally,

$$a \cong t \overset{def}{\Longleftrightarrow} \begin{cases} a = t & \text{if } a \text{ is a leaf node} \\ \exists i \in [1..n] \text{ s.t. } arg_i \cong t & \text{otherwise} \end{cases}$$

where $a = \langle\ \mathrm{id}, \mathsf{type}, arg_1, \ldots, arg_n\ \rangle$. We also assume implicit binding of tuple arguments for this operator, e.g. $t' \cong \mathsf{disj}(a, b)$ where $t'$ is the tuple tree shown in Figure 6.9 (i.e., the *id* of its root dept tuple), is true and binds $a$ to $e_2$ and $b$ to $e_3$.

Notice also that Algorithm 6 uses the $\cong$ operator when looking for source states, but the normal equality operator $=$ when examining conditions. This is important when being in a certain state is only part of a complex condition, e.g.

When the system stays in alarm mode for more than 5 minutes ...

**Algorithm 6** The *Automata* modeler.

```
for all xdep(E, A) ∈ 𝒯 do
  s := ∅; d := ∅; c := ∅ {s source states, d destination states, c conditions}
  for all e ∈ E do
    if e ≅ testmode(obj, mode) then
      s := s ∪ {(obj, mode)}
    end if
    if e ≠ testmode(obj, mode) then
      c := c ∪ {e}
    end if
  end for
  for all a ∈ A do
    if a ≅ setmode(obj, mode) then
      d := d ∪ {(obj, mode)}
    end if
  end for
  if d ≠ ∅ then
    if s ≠ ∅ then
      add automatatrans1(s, c, d)
    else
      add automatatransall(c, d)
    end if
  end if
end for
```

In such cases, both the source state "alarm mode" has to be identified, and the full "system is in alarm mode for more than 5 minutes" has to be taken as an enabling condition.

The *Automata* modeler also collects all the states mentioned in conjunction with each object in testmode (observing state), setmode (changing state), initmode (initial state), and modeset (state declaration) tuples, and builds the

$$\mathsf{automatastates}(obj, \{mode_1, \ldots, mode_n\})$$

model describing the set of all possible states for each object $obj$.

The associate projector provides a graphical representation for the FSA behaviour of all objects with mutable state. This representation is a direct mapping of the automatatrans1 and automatatransall models, where the unparsing of the conditions for each tuple is used as a label for the corresponding arc. Figure 6.11 (a screenshot from CIRCE) shows the FSA model for the following requirements:

1. When system is in normal mode, if current pressure is greater than the given threshold, the system switches to alert mode.
2. If the system is in alert mode for 5 minutes or more, the system issues a "Pressure warning" alarm, and it switches to alarm mode.
3. Whenever pressure becomes lower than the given threshold, the system switches to normal mode.

The *BadAutomata* modeler checks an FSA model against a number of validation properties. Among them:

- an object with mutable state must have at least two states; formally

$$\forall \mathsf{automatastates}(o, M) \in \mathcal{T}, \#M \geq 2$$

- a transition must have only one target state; formally

$$\forall \mathsf{automatatrans1}(s, c, d) \in \mathcal{T}, \#d = 1$$

$$\forall \mathsf{automatatransall}(c, d) \in \mathcal{T}, \#d = 1$$

- transitions must be from states of an object to states of the same object; formally

$$\forall \mathsf{autotamtatrans1}(s, c, \{(o, m)\}) \in \mathcal{T}, \forall (o', m') \in s, o' = o$$

- all states must be reachable from some other state; formally

$$\begin{gathered}\forall \mathsf{automatastates}(o, M) \in \mathcal{T}, \forall m \in M, \\ \exists \mathsf{automatatrans1}(s, c, \{(o, m)\}) \in \mathcal{T} \quad \vee \\ \exists \mathsf{automatatransall}(c, \{(o, m)\}) \in \mathcal{T}\end{gathered}$$

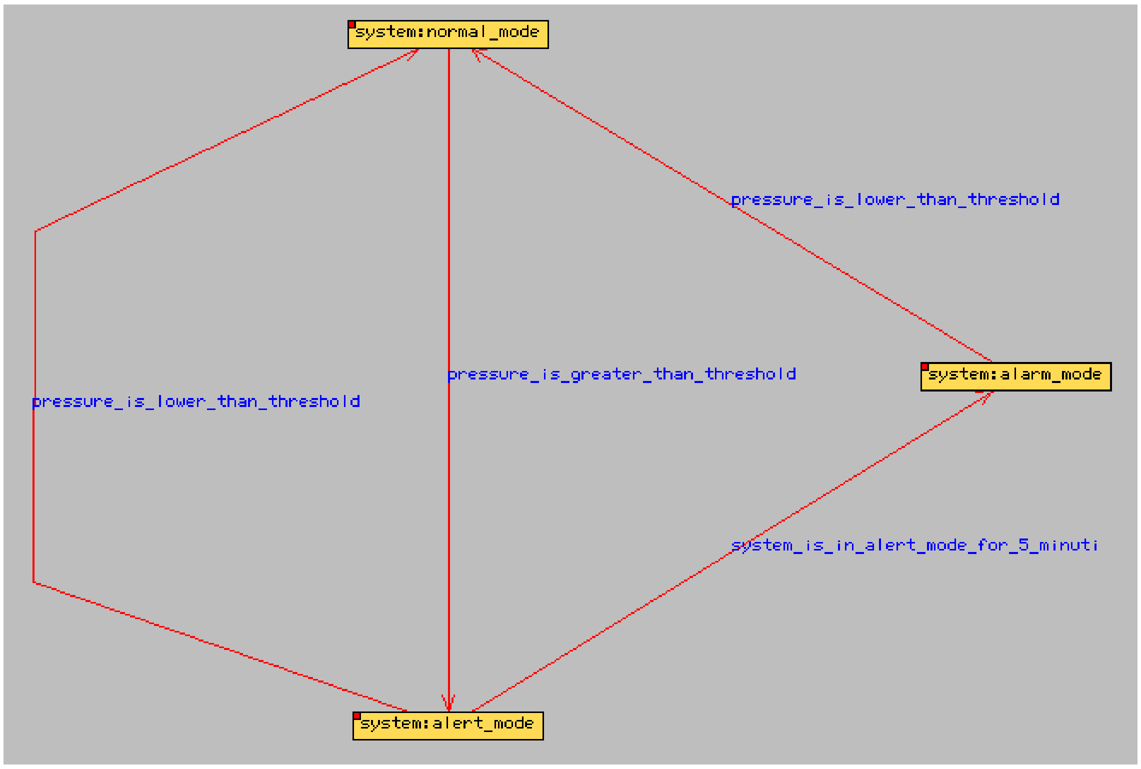



Figure 6.11: Sample FSA model.

- the automata directed graph $G_o$ for object $o$ must be connected;[13] formally

$$\forall \mathsf{automatastates}(o, M) \in \mathcal{T}, \forall m, m' \in M,$$
$$\exists \text{ a directed path } p \text{ from } m \text{ to } m' \text{ in } G_o$$

- if an explicit state declaration was provided, it must be correct and complete, or, in other words, it must include all and only the states for which transitions have been described; formally

$$\forall \mathsf{modeset}(o, M), \exists \mathsf{automatastates}(o, M') \wedge M = M'$$

The outcome of the validation of these properties is recorded in $\mathsf{badautomata}x$ tuples, with $x$ indicating the type of the error, and can be browsed directly by the user via a textual projector.

## 6.4 Document models

The large family of document models are not concerned with the system described in a requirement document, but with the requirement document *per se*. As such they do not rely on specific sentence frames to identify properties of the system, but rather consider the document as a set of statements (term sequences) and associate parse trees (sets of tuples).

Document models include the glossary and requirements input by the user, as well as reports on the outcome of the parsing process and usage statistic for terms defined in the glossary.

They also include models that present or rearrange the structure of the document, as well as some purely linguistic validation check.

### 6.4.1 Parsing and tags

A number of models collect information on the outcome of the parsing process. These models use service tuples output by Cico that describe each parse step, including dropped terms (rests), rules applied and statistics on parsing time, and generally do not build models on their own.

The *Parsing report* view shows a linear version of the parse tree for each requirement. Each node is numbered, and the "@$n$" notation is used to reference other nodes. Moreover, each requirement is shown both in its original form and in canonical form. To enhance recognizability of parsing problems, color is used to attract the attention of the requirement writer on problem spots. For example, the parsing report for the requirements on page 107 looks similar to Figure 6.12.

[13] This property may be relaxed when an object has several sets of orthogonal states, e.g. "normal mode" vs. "in alarm", *and* "working correctly" vs. "failure occurred". In Circe, similar cases are brought to the attention of the requirements writer to ensure their correctness.

1. **When the system is in normal mode, if current pressure is greater than the given threshold, the system switches to alert mode.**

```
WHEN system/SYS/IN/OUT/ELAB/HASSTATE IS IN normal_mode/MODE IF pressure/INF
is_greater_than/RELOP given threshold/INF/BLTIN, system/SYS/IN/OUT/ELAB/HASSTATE
SWITCHES TO alert_mode/MODE.
```

```
   (1) system IS IN normal_mode
   (4) system SWITCHES TO alarm_mode
   (7) pressure is_greater_than threshold
(rest) given
  (10) IF @7 @4
  (13) WHEN @1 @10
```

2. **If the system is in alert mode for 5 minutes or more, the system issues a ”Pressure warning” alarm, and it switches to alarm mode.**

```
IF system/SYS/IN/OUT/ELAB/HASSTATE IS IN alert_mode/MODE FOR 5/NUM MINUTES or
more, system/SYS/IN/OUT/ELAB/HASSTATE ISSUES pressure_warning_alarm/INF/BLTIN
AND IT/< SWITCHES TO alarm_mode.
```

```
  (19) system IS IN alert_mode
  (22) system ISSUES pressure_warning_alarm
  (25) system SWITCHES TO alarm_mode
  (28) @22 AND @25
  (31) 5 MINUTES
  (34) @19 FOR @31
  (37) IF @34 @28
(rest) or more
```

3. **Whenever pressure becomes lower than the given threshold, the system switches to normal mode.**

```
IF pressure/INF is_lower_than/RELOP given threshold/INF/BLTIN
system/SYS/IN/OUT/ELAB/HASSTATE SWITCHES TO normal_mode/MODE.
```

```
  (43) system SWITCHES TO normal_mode
  (46) pressure is_lower_than threshold
(rest) given
  (49) IF @46 @43
```

Figure 6.12: The *Parsing report* view.

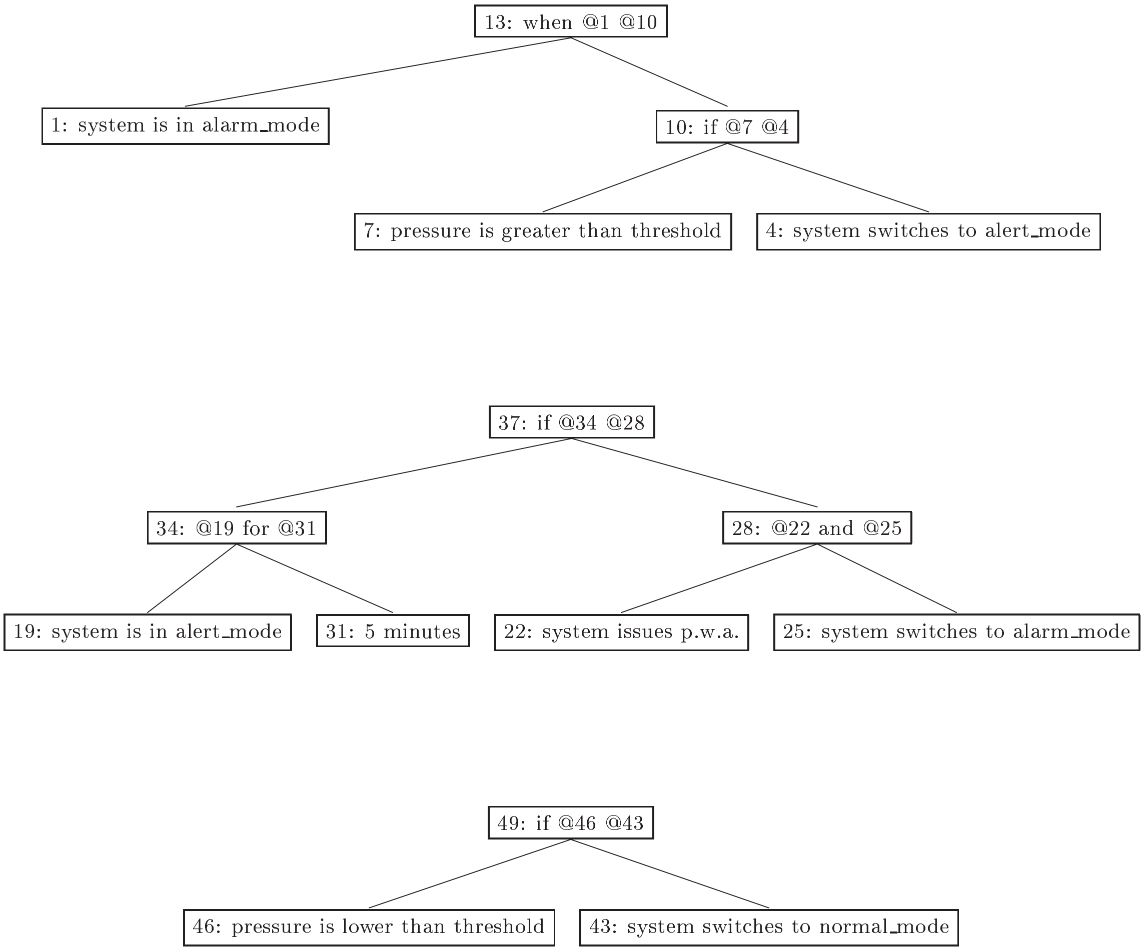



Figure 6.13: The *Syntax trees* view.

Parsing trees can also be observed in graphical form through the *Syntax trees* view (Figure 6.13), although for large documents filtering on specific requirements may be necessary to avoid excessive clutter of the display. A more technical-oriented graphical view on the parsing process is also provided, and has been used in this chapter in Figures 6.1, 6.7, 6.9, etc.

Three views in this family are particularly concerned with glossary terms. They are the *Speculative tagging* view, that shows the user which tags are being proposed for each term by Cico's speculative tagging (detailing for each proposed tag the requirements that suggest it), the *Residual terms* view, that collects all rests found in the parsing process for possible inclusion in the user glossary, and the *Term usage* view, that counts and shows graphically the occurrences in the requirements for each term defined by the user glossary (highlighting unused terms for possible removal from the glossary).

The *Abstraction finding* view, based on the `abstfinder` tool described in [49], presents to the user possible abstractions found in unparsed material. *Abstractions* are, in this context, sets of frequently co-occurring sentence fragments. Differing from [49], where fragments are selected on a pure lexical basis (i.e., arbitrary string of characters), we define sentence fragments as sub-sequences of terms from the canonical form of a requirement. Abstraction finding is particularly useful in the first stages of requirements elicitation, when most requirements are not parsed correctly. Abstractions found can be used to identify domain concepts and progressively refine glossary and requirements until all requirements are parsed correctly (thus resulting in an empty set of residual abstractions).

We also include in this family the *Computation time* service view. It displays in histogram form the time spent by CICO in parsing each requirement. Besides collecting statistics on the performance of the parser, this view may be useful to identify (and possibly split) particularly complex requirements that cause excessive increases in parsing time.

### 6.4.2 Structure

In the document model adopted by CIRCE, the *structure* of a requirements document is simply given by the ordering of the requirements contained therein. CIRCE encourages writing self-contained requirements, that can be completely understood without reference to surrounding text.[14] Self-contained requirements allow CIRCE to quote a requirement when needed without having to place the right context around it, as in many validation views. This also means that CIRCE's interpretation of a requirements document is not affected by the ordering of requirements in the document, nor by the ordering of entries, tags and synonyms in the glossary.

However, human understanding of a document can be (and usually is) greatly affected by the order of presentation. Structure is thus an important concern that is addressed by several modelers and projectors.

So far, we have assumed a requirements document to be a *set* of requirements (Section 5.2.2), and a glossary to be a *set* of definition entries (Section 5.2.3). In the following, we will instead model the structure of both documents as a (possibly nested) *sequence* of requirements or glossary entries, respectively (notice that in sequences elements can be repeated). Formally, we model a structured document $d$ as a pair $(title, \langle e_1, \ldots, e_n \rangle)$ where each $e_i$ can be either a basic document element (a requirement or a glossary entry, depending on the kind of document we are treating) or a sub-document (i.e., a section, with the given title and content).

Given a requirement document $\mathcal{R} = \{r_1, \ldots, r_n\}$, and a glossary $\mathcal{G} = \{e_1, \ldots, e_m\}$,

[14] This impacts several areas in CIRCE's design. For example, encouraging self-containment is the reason for limiting pronoun references in CICO to a single statement, as discussed in Section 5.4.2.

where $e_i = (d_i, s)$, the *GlossUse* modeler builds the auxiliary uglossinvindex model, defined as follows:

$$\mathsf{uglossinvindex}(d_i) = \{r \in \mathcal{R} \mid d_i \text{ occurs in } r_{|\mathcal{G}}\}$$

The *Inverted index* projector simply reorders the requirements by grouping all the requirements according to the terms they reference. Formally, given $\mathcal{R}$ and $\mathcal{G}$ as above, the projector builds a structured version of the requirements document given by

$$\mathcal{R}' = \langle S_1, \ldots, S_m \rangle$$

where each section $S_i$ is defined as

$$S_i = (d_i, \mathsf{uglossinvindex}(d_i))$$

This view is particularly useful to rapidly find all the requirements related to a certain domain entity. A similar view, produced by the *Ordered glossary* projector, sorts the glossary by grouping terms according to their tags. Formally, the projector builds a structured glossary given by

$$\mathcal{G}' = \langle S_1, \ldots, S_n, S_{others} \rangle$$

where each section $S_i$ collects all the terms from $\mathcal{G}$ that have at least a tag in group $i$, i.e.,[15]

$$S_i = (n_i, \{d \mid (d, s) \in \mathcal{G} \wedge \mathsf{tags}(d) \cap \mathrm{tgrp}(n_i) \neq \emptyset\})$$

where the names $n_i$ and the associate set of tags $\mathrm{tgrp}(n_i)$ are given in Table 6.7. The special section $S_{others}$ collects uncategorized terms, i.e.

$$S_{others} = (\text{“Others”}, \{d \mid (d, s) \in \mathcal{G} \wedge \forall n_i, \mathsf{tags}(d) \cap \mathrm{tgrp}(n_i) = \emptyset\})$$

As for the ordered requirements document, grouping substantially improves the requirements writer's ability to spot wrong definitions of terms in the glossary. Moreover, the uglossinvindex model is used by the *Ordered glossary* projector to highlight (by using a different color) glossary entries that are never used in the requirements (and can be removed), thus favouring a stricter adherence of the glossary to the relevant domain.

The models and views we examined so far re-group and re-order the contents of the requirements document and of glossaries (both user and system ones) based on simple occurrence tests. To obtain a better handling of requirements documents, however, we need more refined relationships among requirements. Building such a

[15] Although section contents are specified as a set here, the actual view presents a sequence of lexicographically-ordered entries.

| **Group name** ($n_i$) | **Group tags** ($\mathrm{tgrp}(n_i)$) |
|---|---|
| Communicating systems | /IN /OUT /IRQ /RIRQ |
| Active systems | /ELAB /HASSTATE |
| Information content | /INF /BLTIN /SHARED |
| Attributes | /ATTR |
| Ordinals | /NUM |
| Relationships and operators | /REL /RELOP /BINOP /UNOP |
| Events and actions | /EVT /ACT |
| States and modes | /MODE |
| Parts of the system | /SYS |
| Time-related | /TSPAN /PIT /TINTERV |

Table 6.7: Groups for the *Ordered glossary* view.

relationship is the purpose of the *ReqCorr* modeler. The reqcorr model it synthesizes from the uglossinvindex model provides a similarity measure, based on cosine correlation, between each pair of requirements, computed as follows:

$$\mathsf{reqcorr}(a,b) = \frac{\sum_{(k,s)\in\mathcal{G}}(f_a(k) * f_b(k))}{\sqrt{(\sum_{(k,s)\in\mathcal{G}} f_a(k)^2) * (\sum_{(k,s)\in\mathcal{G}} f_b(k)^2)}}$$

where $a$ and $b$ are requirements, and $f_x(t)$ is the frequency of occurrence of term $t$ in a glossary-mapped requirement $x$, i.e.

$$f_{\langle x_1,\ldots,x_n\rangle}(t) = \frac{\#\{i \mid x_i = t\}}{\#\{i \mid \exists (t',s') \in \mathcal{G} \text{ s.t. } x_i = t'\}}$$

The reqcorr model can be used in a number of ways to analyze the structure and to re-structure a requirement document. *Correlation graph* and *Correlation table* are two projectors that present the reqcorr model directly to the user, for inspection and analysis.

The former displays correlation as a matrix in which each cell $(i,j)$ is assigned a colour depending on the value of $\mathsf{reqcorr}(r_i, r_j)$ (Figure 6.14). Such a view can be used to give a rapid assessment of the homogeneity of the structure of the document. Ideally, each section (i.e., each block of consecutive statements in our linear model of requirements document) should exhibit a high degree of internal homogeneity, and be clearly separated from other sections. In the *Correlation table* view, this corresponds to having an approximately block diagonal matrix. In Figure 6.14, for example, requirements 15–27 constitute a block, partially overlapping with block 24–34; other, less defined block can be observed in requirements 1–5 and 7–9. On the contrary, isolated stripes hint at a bad placement for a requirement, as is the case for requirement 3 (it could join block 7–9) and for requirement 5 (it could join block 24–34). Also, requirements 24 and 26 should be separated from requirements

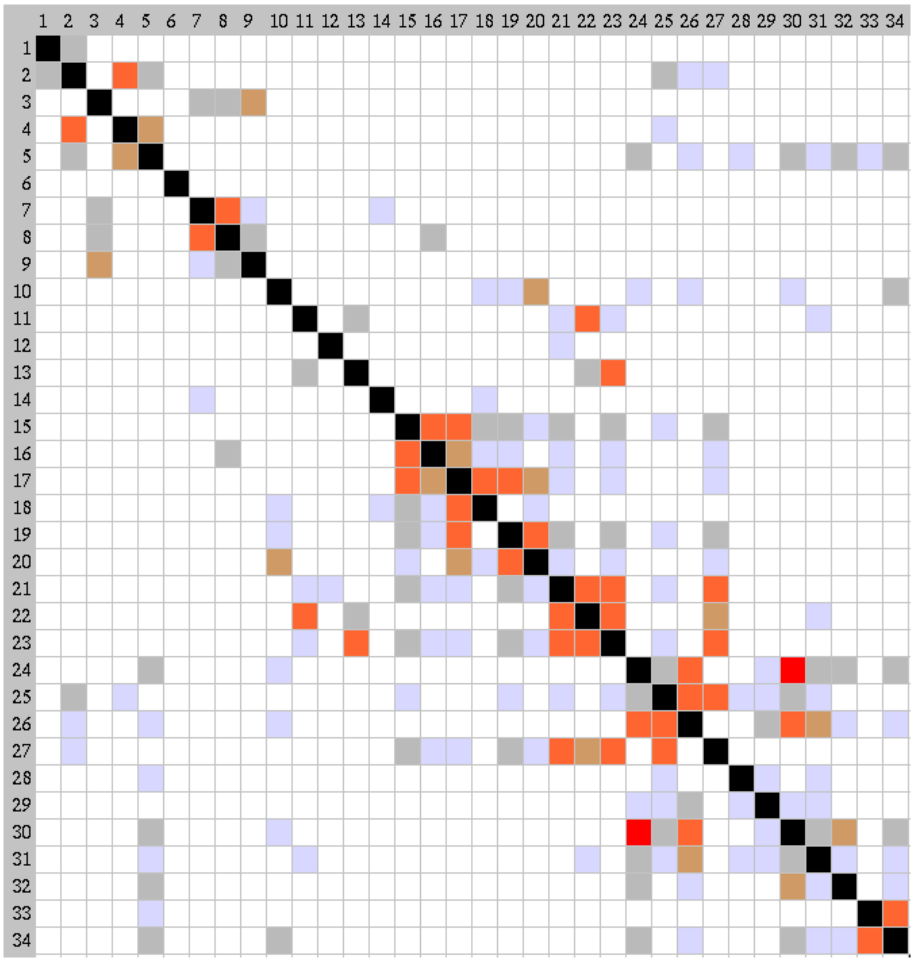



Figure 6.14: *Correlation table* view.

23, 25 and 27, and moved after requirement 27, thus reducing the overlap between the bottom blocks.

The *Correlation graph* view displays the same information in graph form; each node represent a requirement, while edges represent a correlation coefficient greater than a given user-selectable threshold. Also in this case, the automatic graph layout algorithm used by CIRCE's interaction environment helps in identifying clusters of related requirements.

A better systematic treatment to the problem of changing the structure of a document is given by the reqcluster model. Given the reqcorr model, the *Clusterization* modeler tries to identify hierarchical clusters of requirements that could form sections, subsections, etc. in a restructured version of the source document.

A *correlation graph* $(N, E)$ is simply an undirected complete graph whose nodes are requirements, and whose edges have an associate numeric value (the correlation coefficient). A *$\sigma$-reduced* correlation graph is a correlation graph $(N, E')$ where

$$E' = \{(a, b) \mid \mathsf{reqcorr}(a, b) \geq \sigma\}$$

The clustering algorithm implemented by the *Clusterization* modeler selects clusters in a $\sigma$-reduced graph by identifying potential heads (requirements with the highest degree) and by clustering all their neighbours together. The head and incident edges are then removed from the graph and the process is repeated until all nodes have been assigned to a cluster (see Algorithm 7).

**Algorithm 7** Cluster($\mathcal{R}$)

Let $G = (N, E)$ be a $\sigma$-reduced correlation graph for $\mathcal{R}$
$M := N$
**while** $M \neq \emptyset$ **do**
  $h := \text{maxarg}_{n \in N}\text{degree}(n)$
  $P := \{h\} \cup \{n \in N \mid (h, n) \in E\}$
  output cluster $(h, P)$
  $N := N \setminus \{h\}$
  $M := M \setminus P$
  $E := E \setminus \{(h, p) \mid p \in P\}$
**end while**

Notice that nodes assigned to a cluster are left in the graph, and may later become heads themselves of other clusters. This induces a hierarchical structure on clusters, and a single requirement or an entire cluster can appear as member of more than one cluster.

The clusters identified by Algorithm 7 constitute the reqcluster model, that is used by the *Restructure* projector to present a restructured version of the document. Each requirement $r$ or cluster $c = (h, s)$ of a reqcluster model $R$ is assigned as *principal* cluster the one whose head has the highest correlation coefficient with $e$, i.e.

$$\begin{aligned}\text{principal}(r) &= \text{maxarg}_{(h',s')\in R}\mathsf{reqcorr}(r,h')\\ \text{principal}(c) &= \text{maxarg}_{(h',s')\in R}\mathsf{reqcorr}(h,h')\end{aligned}$$

In displaying the document, each structure element (requirement or cluster, representing a section) is shown either at the top element, if it is not member of any other cluster, or inside the section corresponding to its principal. Other occurrences are instead substituted by "(see also…)" links.

Each section is then assigned a title on the basis of the terms that are most frequent among its members (the $f_x(t)$ measure is used again for this purpose), and the structure is flattened (i.e., deeply embedded sections are expanded in line in the body of their principal) or expanded (by splitting an excessively long section $S$ through repeated application of Cluster($S$) with progressively higher $\sigma$ parameter) according to user-selectable limits on the depth of the structure ($\delta$) and on the length of sections ($\lambda$).

Although no formal assessment has been conducted so far, we have found the following values of the parameters to produce reasonable results in common usage:

$$\sigma = 0.6 \qquad \delta = 3 \qquad \lambda = 9$$

Notice that our definition of correlation only considers terms defined in user glossaries to estimate similarity between requirements. This choice is appropriate when the desired structure for the document is modeled on the *domain* structure of the problem. Alternatively, all terms from both user and system glossaries can be considered, and in this case similarity in functionality is also taken into account (e.g., all requirements describing computations could be collected in a cluster, regardless of the agent, results and parameters of the computations).

While a carefully hand-crafted structure for a requirements document is usually preferable to a synthesized one, automatic restructuring can be an invaluable help when the document is obtained by merging different documents, possibly produced by different users working on the same project. The new structure can then be used to re-assign homogeneous sections to the participants, or to consolidate in a single document, to be refined later, partial contributions by different people.

### 6.4.3 Validation

A few models perform validation checks that are independent from a particular model, and are discussed in this section.

The *IncompatibleTags* modeler assures that terms are not given incoherent descriptions by assigning to them incompatible (or plainly wrong) tags. For example, `/NUM` and `/ELAB` are considered incompatible tags, since a number cannot possibly perform computations.

The following groups of tags are considered *compatible* each other:

- `/INF`, `/NUM`, `/BLTIN`, and `/SHARED`;
- `/ACT` and `/EVT`;
- `/IN`, `/OUT`, `/ELAB`, `/IRQ`, `/RIRQ`, and `/SYS`;
- `/PIT` and `/EVT`;
- `/EVT`, `/TINTERV`, and `/TSPAN`;

Moreover, `/BLTIN` and `/SHARED` require that the term also has an `/INF` tag, and any tag unknown to Circe is marked as such. Despite its simplicity, the badtags model produced by *IncompatibleTags*, completed with some guidance, has proven to be extremely useful in assisting unexperienced requirement writers during the first stages of domain description.

Two other modelers validate requirements against *redundancy* and *non-atomicity*. In the first case, requirements are identified whose entire content is already known from other requirements; such requirements can be removed from the requirements document without losing any information (and usually improving readability due to the reduced size of the document). In the second case, requirements expressing completely separated concepts are identified, and proposed for splitting in two separate requirements, again improving readability and analysis capabilities.

In both cases, the auxiliary model arroot is used. The corresponding *ReqsRoots* modeler uses service tuples produced by Cico to obtain, for each source requirement $r$, the set of tuples $T(r)$ produced during the parsing of $r$. The model is then computed according to the following definition:

$$\mathsf{arroot}(r) = \{\langle id, \ldots \rangle \in T(r) \mid \nexists j, \langle id', t, a_1, \ldots, a_m \rangle \in T(r) \text{ s.t. } a_j = @id\}$$

In other terms, $\mathsf{arroot}(r)$ is the set of all the root tuples in the forest of parse trees for requirement $r$. Non atomic requirements are thus identified by the model

$$\mathsf{nonatomic}(r) \quad \stackrel{def}{\Longleftrightarrow} \quad \#\mathsf{arroot}(r) > 1$$

that is reported to the user, together with the unparsing of each subtree, for correction and possibly splitting. For example, consider the following requirement $r$:

1. When the system, that is initially in normal mode, receives a bad command from the client, it switches to error mode.

whose canonical form is

```
WHEN system/HASTATE/ELAB/IN that/< IS INITIALLY IN
normal_mode/MODE RECEIVES bad_command/INF FROM client/OUT it/<
SWITCHES TO error_mode/MODE
```

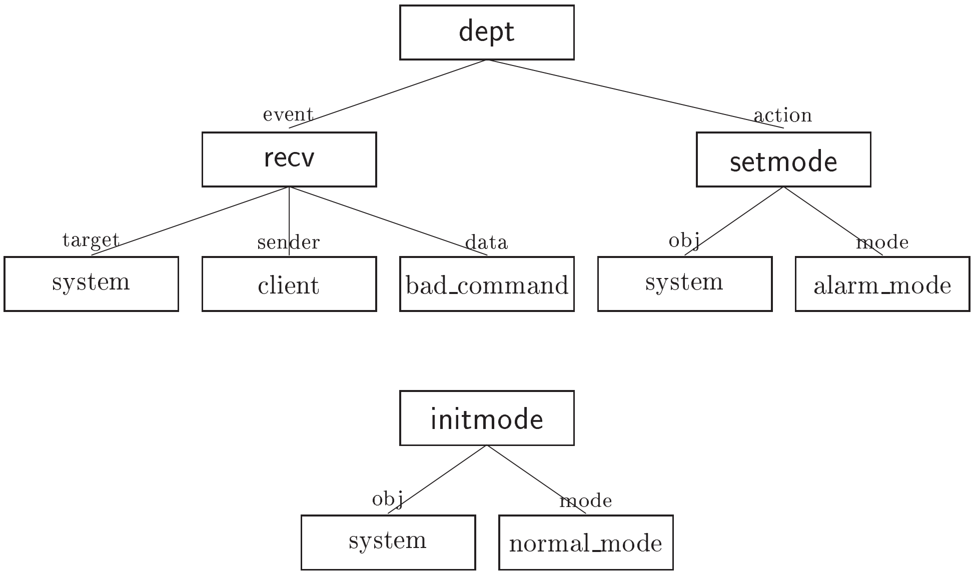



Figure 6.15: Parsing trees for a non-atomic requirement.

Parsing this requirement would produce the parse trees shown in Figure 6.15, where it is evident that the requirement is non atomic. Indeed, the projector for nonatomic models would inform the user that our requirement could be split in two atomic pieces:

1. The system is initially in normal mode.
2. When the system receives a bad command from the client, it switches to error mode.

Redundant requirements are also identified by using the arroot model. The *RedundReq* modeler compares each atomic parse tree (i.e., each parse tree rooted in a member of $\mathsf{arroot}(r)$) with all the other subtrees in the forest of parse trees for the document. Two subtrees are declared equivalent (denoted by $\approxeq$) when either they are identical, apart from renumbering of *id*s, or when some domain-specific auxiliary modeler declares them to be equal. For example, a modeler in the data flow family declares that

$$\mathsf{recv}(X, Y, Z) \approxeq \mathsf{send}(Y, X, Z)$$

while others declare conjunctive (resp. disjunctive) lists containing the same terms to be equivalent regardless of order of occurrence of the terms (except in cases where order matters, for example in a conjunctive list representing an ordered sequence of actions). Domain-specific equivalence is indicated through $\mathsf{eqtree}(a, b)$ tuples, meaning that the (sub)trees rooted at $a$ and $b$, respectively, should be treated as equivalent. Formally, given $a$ and $b$ that are either terms or tuple identifiers (*tid*s; in this case, we assume the correspondent tuples are $\langle a, t_a, a_1, \ldots, a_n\rangle$ and $\langle b, t_b, b_1, \ldots, b_m\rangle$, respectively), we define

$$a \approxeq b \quad \overset{def}{\Longleftrightarrow} \quad \begin{cases} a = b & \text{if } a \text{ and } b \text{ are terms} \\ true & \text{if } a \text{ and } b \text{ are } tids \wedge \\ & \mathsf{eqtree}(a,b) \\ t_a = t_b \wedge a_1 \approxeq b_1 \wedge \ldots \wedge a_n \approxeq b_m & \text{if } a \text{ and } b \text{ are } tids \wedge \\ & \neg\mathsf{eqtree}(a,b) \wedge n = m \\ false & \text{otherwise} \end{cases}$$

The $\approxeq$ relational operator can be used to identify all the subtrees in a repository $\mathcal{T}$ equivalent to a given tree $t$:

$$\text{equiv}_{\mathcal{T}}(t) = \{t' \in \mathcal{T} \mid t \approxeq t' \wedge t \neq t'\}$$

Completely equivalent requirements (or atomic fragments of requirements, see the discussion of nonatomic above) are thus described by the eqreq model as follows:

$$\mathsf{eqreq}(r_1, r_2) \quad \overset{def}{\Longleftrightarrow} \quad \text{isroot}(r_1) \wedge \text{isroot}(r_2) \wedge r_1 \approxeq r_2$$

while redundant requirements are described by the redundreq model:

$$\mathsf{redundreq}(r_1, r_2) \quad \overset{def}{\Longleftrightarrow} \quad \text{isroot}(r_1) \wedge \neg\text{isroot}(r_2) \wedge r_1 \approxeq r_2$$

where the auxiliary function isroot is defined as follows

$$\text{isroot}(r_i) \quad \overset{def}{\Longleftrightarrow} \quad \exists r \in \mathcal{R} \text{ s.t. } r_i \in \mathsf{arroot}(r)$$

Thus, $\mathsf{redundreq}(r_1, r_2)$ indicates that all the information that can be extracted from $r_1$ can also be extracted from $r_2$ (that adds more information on its own). The *RedundReq* projector presents this information to the user, proposing the deletion of all but one of the redundant requirements (for eqreq), or the deletion of contained requirements (for redundreq). For example, consider the following requirements:

- Temperature is an attribute of a boiler.
- When the temperature of a boiler exceeds the limit, ...

The first requirement declares an attribute and produces an attr tuple (see Table 6.1), but so does the second one, even if in this case the reference to the temperature is in the context of an access to read the current value of the attribute. Requirement 1 would be declared redundant with respect to requirement 2, and in fact the information that a boiler has an attribute called "temperature" can be obtained by requirement 2 alone; requirement 1 can be deleted without any loss of information.

## 6.5 Metrication modelers

A number of modelers are devoted to the collection of measures on the models produced by other modelers. Metric data collected can be used simply to characterize a product in its current state or the process producing it, but also to verify that the product or process complies with a defined quality model. Circe does not define a quality model on its own, in keeping with our belief that quality models depend on human, organizational, normative and environmental conditions, and are thus organization-specific (and at times project-specific). A quality model such as our proposal in [37] can however be added to the environment by writing an appropriate modeler and projector.

In this section, we will concentrate on *metrication* (i.e., the collection of values for attributes of a measured entity), and not on the interpretation of the measures collected, that is a matter left to specific quality models and desired quality profiles. As usual, we will distinguish metrication modelers according to their use of document models, of system models, and of process models.

### 6.5.1 Document measures

Measures collected on the requirements document are essentially counts of various textual and structural items. Several measures of the *size* of a requirements document are provided:

- the *number of natural language requirements* and *total length in words of the document* give a rough estimate of the size of a requirements document as a printed artifact, that can be used, for example, to estimate the time needed to read it;
- the *number of atomic requirements* (i.e., number of nodes in parse trees for the requirements) and the *cumulative length in terms of canonised natural language requirements* both give a more accurate estimate of the size of a requirement document once lexical and syntactic variations are discarded.
- the *number of distinct tuple types* produced by the parsing process provides an indication on the diversity of matters treated in the document;
- the *total number of terms in the glossary* (including synonyms) and the *number of base terms in the glossary* estimate the textual and effective complexity of the domain, interpreted as the number of different entities playing a role in the document;
- the *number of* `/SYS`*-tagged base terms in the glossary* provides an indication on the number of different entities the document will presumably focus on.

Other measures collected about document attributes include classical readability formulas (e.g., Flesch readability index [45] and its variants for different languages) and the homogeneity of structural elements like chapters and sections, measured through the reqcorr model of Section 6.4.2.

### 6.5.2 System measures

System measures are obtained by collecting metric data on the attributes of models of the system described in the requirements document. Most measures can be computed by simply inspecting the relevant models. For example, the following counts are taken from the entity models:

- the *number of attributes* for each entity;
- the *total number of relationships* in which an object participates;
- the *number of distinct relationships* in the E-R model;
- the *number of disjoint components* in the E-R model.

Estimates of the complexity of the domain description part of a requirements document are computed as derived measures from the counts collected from the er$xx$ and attr models. Other metrication modelers collect measures on the is-a and part-of models (e.g., the number of descendant and ancestors for each entity in the is-a model) that can be used in a quality model to check whether the requirements comply with chosen standards (e.g., no multiple inheritance in descriptions).

The *MFeP* metrication modeler is responsible for computing the *Modified Feature Points* functional metric model. Feature Points are an enhanced version of the Function Points metric [64, 65] that take into account the complexity of algorithms. The mfepcount model records, for each active entity mentioned in the requirements, the total number of inputs and outputs,[16] the total number of distinct datastores, and the total algorithmic complexity of the entity, estimated through the number of parameters and of results computed. Formally,

$$\mathsf{mfepcount}(e)_{\mathcal{T}} = \langle nin(e,\mathcal{T}), nout(e,\mathcal{T}), nds(e,\mathcal{T}), tac(e,\mathcal{T})\rangle$$

where

$$\begin{aligned} nin(e,\mathcal{T}) &= \sum_{\mathsf{dflow}(s,e,D)\in\mathcal{T}} \#D \\ nout(e,\mathcal{T}) &= \sum_{\mathsf{dflow}(e,r,D)\in\mathcal{T}} \#D \end{aligned}$$

[16]Incoming interrupt requests and server queries are counted as inputs, while outcoming interrupt requests and answers to server queries are counted as outputs.

$$\begin{aligned} tac(e, \mathcal{T}) &= \sum_{\mathsf{elab}(e,P,R) \in \mathcal{T}} 1 + \log \#P + \log \#R \\ nds(e, \mathcal{T}) &= \# \bigcup D \text{ s.t. } (\ \mathsf{dflow}(s, e, D) \in \mathcal{T} \vee \mathsf{dflow}(e, s, D) \in \mathcal{T} \vee \\ &\qquad\qquad \mathsf{elab}(e, D, R) \in \mathcal{T} \vee \mathsf{elab}(e, P, D) \in \mathcal{T}\ ) \end{aligned}$$

The counts collected in the mfepcount model are used, in a weighted sum according to the Feature Point metrics, to compute the MFeP (functional) score $F_{MFeP}(e, \mathcal{T})$ for each active entity mentioned in the requirements. While the exact coefficients used can be adjusted according to the class of system being considered (i.e., embedded software *vs.* management information system), the following definition is used by default in CIRCE:

$$F_{MFeP}(e, \mathcal{T}) = 4 * nin(e, \mathcal{T}) + 5 * nout(e, \mathcal{T}) + 3 * tac(e, \mathcal{T}) + 1.5 * nds(e, \mathcal{T})$$

The *Functional score* projector presents to the user a table showing for each entity its MFeP score and the projected effort for implementing the component in different languages, estimated on the basis of empirical historical data [65].

A slight variation of the mfepcount model is used to compute the *cost of presence* of each requirement. The cost of presence of a requirement $r$ is defined as the difference between the total functional score of the system, as described by the requirements document, and the total functional score that the system would have if $r$ is removed from the document. Formally,

$$\mathsf{costofp}(r) = \sum_e F_{MFeP}(e, \mathcal{T}) - \sum_e F_{MFeP}(e, \mathcal{T} \setminus T(r))$$

where $T(r)$ is the set of tuples derived from requirement $r$. Notice that the flows and computations described in $r$ could also be described in other requirements, so it is possible that each single requirement in a group have a cost of presence of 0 (i.e., dropping it would not reduce the functional score for the system), while the entire group as a whole have a cost of presence greater than 0. Cost of presence analysis can be extended by limiting the sums to a single active entity or to a group of entities (e.g., an entire subsystem), or by applying it to arbitrary sets of requirements (e.g., whole sections of the document) instead of removing a single requirement at a time.

A related projector presents to the user a list of requirements prioritised according to their cost of presence. The requirement writer can then evaluate the cost/benefits ratio for each requirement, for example to discard costly but optional requirements if needed.

A few of CIRCE's metrication modelers work on the behaviour models we discussed in Section 6.3. Complexity of behaviour is characterized by different measures:

- the *total number of decision points* for each entity, i.e., the number of `depc` and `dept` tuples that have the given entity as initiator of the action part;
- the *total number of different behaviours* possible for an entity, obtained by counting the number of different `xdep` tuples that have the given entity as initiator;
- the *cumulative size of conditions* to verify, i.e., the sum of the cardinalities of the events sets in all the `xdep` tuples that have the given entity as initiator;
- the cumulative *volume of decision trees*, computed by multiplying the maximum nesting level of a `dep`$x$ tuple in a decision tree for a given entity by its maximum breadth.

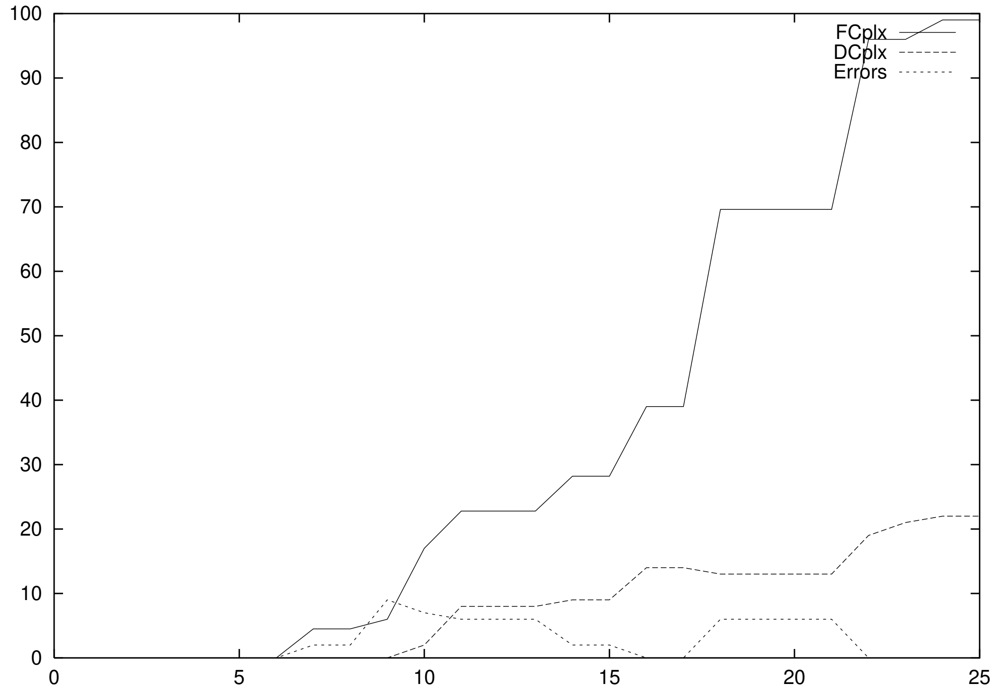
An associated projector presents these data to the user by listing them in tabular format. Derived measures similar to the cost of presence measure presented above are also calculated, and the same considerations apply.

### 6.5.3 Process measures

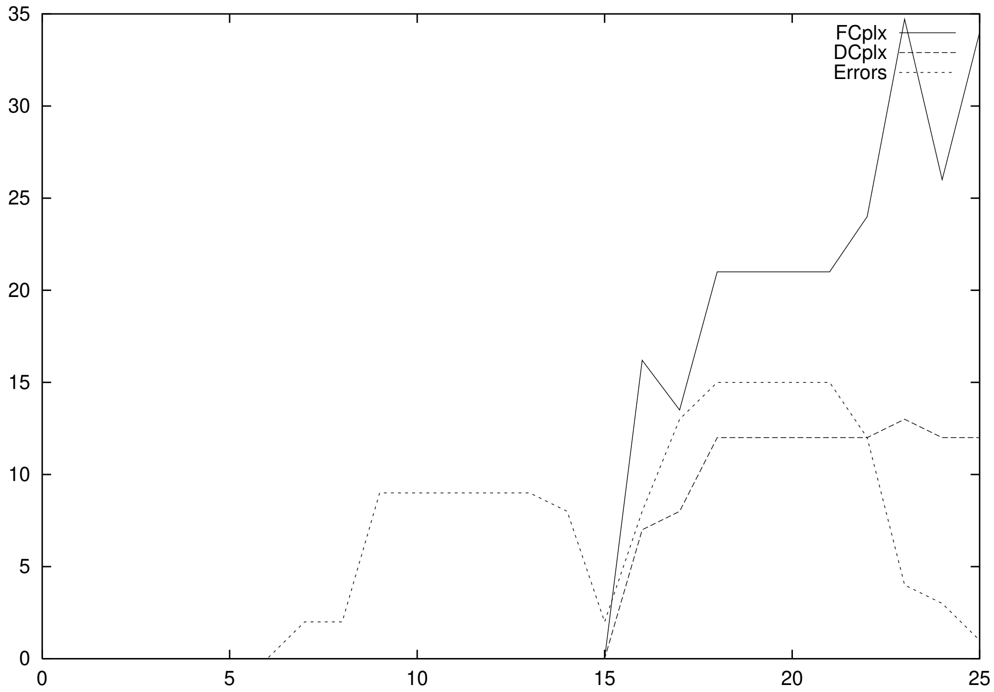
Circe supports controlling and monitoring the requirement writing process in itself by projecting product attributes (i.e., document and system metrics) on the time axis. This is obtained by considering the RCS repositories into which Circe stores each version of the requirements as syntactic repositories for specialized modelers. In other terms, it is possible to observe the evolution of all the measures collected by other modelers (and of other characteristics, i.e., the number of violations of validation properties) during the requirement writing process.

For example, Figure 6.16 shows the evolution of three measures during the first three weeks of requirements analysis by two teams working on the same problem. The three measures considered are functional complexity FCplx, computed from the MFeP measures discussed above, dynamic complexity DCplx, computed from the behaviour measures, and the total number of errors identified by Circe's validation modelers.

It can be observed that Figure 6.16(a) depicts a smooth, controlled process. After a first week (probably spent in familiarizing with the environment), the team gradually added description of functionality and behaviour, while keeping errors under control. On the contrary, the team in Figure 6.16(b) spent two weeks in familiarizing with the tools, and only started describing functionality and behaviour on day 15. However, the team did not take time to correct the errors identified by Circe until later in the process, on day 22. In a last-minute effort only a few days before the milestone deadline for the first inspection, the team concentrated in removing the errors, which they did with success. This kind of late debugging activity adversely affected the stability of the requirements, as can be observed in the glitches that characterize the last two days of both FCplx and DCplx. Also




(a)




(b)

Figure 6.16: Traces of two requirements writing processes.

notice that the second team only reached a functional score of 34, a third of the score reached by the first team. The requirements produced by the second team, even if formally correct, are probably too abstract to give solid guidance for the design phase.

Product metrics traces like the one shown above allow project managers to take appropriate actions when a process exhibits erratic, unstable or undesirable behaviour. They also allow us to formally characterize processes according to their features. For example, in [6] a quantitative characterization is proposed based on the *stability* attribute of the requirement process.

While the requirement process can be ideally modelled as the progressive accumulation of information about the system to design, in practice it often has an *impulsive* behaviour. Experience shows that analysts tend to proceed in alternating phases of writing (adding new requirements) and polishing (correcting errors, restructuring descriptions). Stability is a measure of how smoothly these two phases are integrated. It is maximal for the ideal process in which information grows linearly with time (the process in Figure 6.16(a) is a good approximation of the ideal process), and gets worse for highly impulsive processes.

To measure stability, we first consider some measure $F(t)$ of the amount of information contained in the requirements (e.g., FCplx or DCplx) at time $t$. The differential measure $\Delta F(t) = F(t) - F(t-1)$, defined for $t > 0$, indicates, for each instant $t$, the difference in information content with respect to the previous version of the requirements. A frequency analysis (Fourier transform) of $\Delta F(t)$ provides information about the distribution and magnitude of the impulsive phenomena in the analysis activity. We can identify four classes of processes with respect to stability, according to the principal frequency components of $\Delta F(t)$:

I. *low peaks on high frequencies.* Small, continuous changes; the activity is strongly incremental in nature.

II. *low peaks on low frequencies.* Small, infrequent changes; the activity proceeds by relatively minor, well spaced impulses.

III. *high peaks on low frequencies.* The activity proceeds by substantial jumps apart each from the other; the development activity is thus strongly impulsive.

IV. *high peaks on high frequencies.* Frequent modifications of relevant magnitude; the process is characterized by continuous and important upheavals.

Under the hypothesis that stable processes lead to products of predictable quality, we can expect that processes of class I (incremental) produce better requirement documents, while processes of class IV (catastrophic) end up producing documents of dubious quality[17]. Processes of class II (sleepy) and III (impulsive) can be expected

[17] We use the term *dubious* here to mean that in some particular case the quality of the requirements could indeed be reasonable, but there is no way to predict the outcome beforehand: it depends almost entirely on the very last “revolution” in the document.

to produce results of intermediate quality.

This hypothesis can be validated by independent evaluation of the results of processes of the various classes, as described in [6] (see also [2]).

Circe also provides some limited process guidance in a projector called *Pandora* that collects and prioritises violations of the various validation properties we have described in the previous sections. In presenting a summary of the violations, *Pandora* adds explanations, examples and suggestions for possible ways of remedying to the errors. Our experiences with the use of Circe in academic environments confirm that even this limited process guidance is a valuable tool for inexperienced requirements writers.

# Part III

# Case Studies

# Chapter 7

# The Space Station's NCS[1]

Abstract

In this chapter we report on our experiences of using Circe for the partial validation of a fragment of NASA's specification of the Node Control Software of the International Space Station. These experiences support our position that it is feasible and useful to perform automated analysis of requirements expressed in natural language. Indeed we identified a number of errors in our case study that were also independently discovered and corrected by NASA's IV&V Facility in a subsequent version of the same document.

In this experiment we tested the applicability of Circe to a new language (English instead of Italian) and writing style (NASA technical documents), to a new domain (aerospace control systems), and to new models, with different properties.

## 7.1 Purpose of the study

The use of lightweight formal methods has recently received increasing attention in the software development literature [39, 61]. We use here the term "lightweight formal methods" to characterise those methods whose adoption cost is a small fraction of that of the overall RE process, including training, application and computational costs. Lightweight formal methods often perform partial analysis on partial specifications only [34], but on the other hand they do not require a commitment to translate an entire (informal) requirements document into a formal one, nor to maintain formal and informal versions of specifications in parallel [68]. Moreover, as requirements specifications evolve during the early stages of the RE process, lightweight formal methods provide an opportunity for gradually validating requirements, paving the way for later introduction of more exhaustive and rigorous analysis if needed.

[1] The work described in this chapter was conducted jointly with Bashar Nuseibeh, and supported in part by ESPRIT Project 20.800 "RENOIR".

A number of experiences have been reported on the use of lightweight formal methods. These range from their application to the very early stages of a development process (e.g., [49] who use lexical analysis to find abstractions in unstructured and uninterpreted text), to reengineering applications on existing code [86]. The application of such methods to the analysis and validation of NL requirements is particularly appealing, since industrial practice shows that NL requirements are easier to evolve, maintain and discuss with (possibly non-technical) customers. However, it is often very difficult to prove properties such as correctness, consistency and minimality about NL requirements.

In this study we wanted to test two characteristics of environment-supported requirement analysis:

1. the adaptability of Circe to a new domain, to a new style, to a new language, and to new models;

2. the effectiveness of Circe as a lightweight validation tool for preexisting requirements documents.

Regarding the first characteristic, Circe's rules set (originally in Italian) was translated into English, and new rules were developed in order to correctly parse the language used in a requirements document from NASA. Moreover, since the system described in the document covered a domain that was not available in Circe's library at the time, new models had to be developed.

Regarding the second characteristic, a number of properties were automatically checked on the newly-developed models, and the outcome of the validation was compared with that of the independent verification & validation (IV&V) process customarily used by NASA.

This experience is thus representative of the case when Circe or a similar environment is introduced in industrial usage. The case study provides data on the effort required for this introduction, and on its foreseeable effects on the quality of the resulting requirements.

The chapter is structured as follows. We begin by providing some background to our case study, followed by a short presentation of a general framework for validation of natural language requirements. Then we describe the application of our framework to the case study, discuss our findings, and reflect on the lessons we learned.

## 7.2 The case study

We studied a fragment of a NASA Software Requirements Specification (SRS) for the Node Control Software (NCS) on the International Space Station [87]. The choice of this particular document was appealing because we assumed it to be of high quality (being the 12$^{\text{th}}$ release of those requirements, and subject to many inspections and revisions), and because parts of it had already been analyzed using different techniques, in related studies [34, 100, 101].

The document, 250 pages long, is written mainly in narrative English, with several tables and the occasional schematic diagram interspersed in the text. The 3-page fragment we chose to analyze described one of the basic components of the Environmental Control function — Cabin Pressure Monitoring. The NCS continuously monitors the cabin pressure, and issues alarms if the measured pressure exceeds operating limits. This function can be disabled and enabled as part of Fault Detection, Isolation and Recovery (FDIR) procedures.

The document is structured by NCS functions (e.g., Telemetry Control, Environmental Control, Time Management, etc.). Each function is described in terms of individual constituent components (e.g., Environmental Control includes pressure monitoring, air fan control, fire & smoke detection, etc.). Each of these components, in turn, is first introduced in general, narrative terms, and then detailed by describing its inputs, outputs and expected behaviour.

## 7.3 Validating NL requirements

The process we follow for automatic partial validation of NL requirements is structured in a setup phase and a production phase. The setup phase includes the following activities:

1. *Defining a style, a structure and a language for the requirements document.* This step can be meant either normatively, i.e., as the production of a prescriptive style manual for the requirements document (and in this case a syntax-guided editor can be used to support requirements writing), or descriptively, i.e., as an adaptation of the capabilities of a parsing tool to an already existing document written in a defined style.

2. *Selecting desirable properties to check.* Which properties of a certain document or system described in a document are "interesting" depends on the particular context of the analysis. Partial validation is usually acceptable at this stage.

3. *Defining one or more models on which the properties selected in the previous step can be checked.* Properties are always relative to models, i.e., abstractions of the document or of the system described in it, which collect in an analyzable structure the information needed to check the property. For example, a connection property among system components can be checked on a model describing all the communications among system components.

Once the setup phase has been completed, the production phase can be iterated at any stage of development of the requirements — without incurring any significant additional cost, as we will show later. The production phase includes:

4. *Pre-processing the requirements document*, to handle format, structure and typographical details and to translate the requirements document to a canonical form amenable to later processing.

5. *Parsing the NL text of the requirements*, leading to an analyzable representation of the semantic content of the text. Again, parsing can be (and usually is) partial, to help in reducing the cost of the validation, as long as that does not interfere with the collection of the information needed to perform the validation.

6. *Building the models* defined in step 3 above, using the information collected during the parsing process. It is possible to build models of the requirements document (for example, distribution of topics among sections of the document) and of the system described by the requirements (for example, a model of the communication paths in a distributed system).

7. *Checking that the models satisfy chosen properties.* As in the previous step, it is possible to check properties of the document (in our previous example, consistency of topics inside a single section) and of the system (for example, the existence of disjoint components in the communication paths model).

8. *Evaluating findings and revising the requirements specification accordingly.* It is particularly important that the validation checks provide as much detail as possible about the point and the reason of a failure (i.e., on the circumstances in which a validation property was violated). Similarly to the counterexamples provided by some full formal methods, this information helps the requirements engineer to identify and fix errors that cause violations.

This process offers a number of advantages in an industrial setting. Steps 1–3 are reusable across projects, as each organization tends to adopt defined internal standards for document style (step 1) and quality control (steps 2 and 3). Moving these standards into a tool is an effective way to accumulate the organizational knowledge and expertise in a safe and structured way, and to have it applied in a deterministic and reproducible manner during the production phase. Also, steps 4–7 are entirely automatic, leaving step 8 only for the requirements engineer to consider at each iteration.

## 7.4 Application to the case study

In this section we describe the details of our experience with the case study, according to the structure of the process outlined above. Since — due to logistic considerations — we had to work alongside NASA's standard verification and validation process, and not inside it, we only ran a single iteration of our production phase. Also, we analyzed three major revisions of the requirements document in "batch mode". This is not the best possible setting for a RE supporting environment, which is actually better suited for continuous application during requirements evolution between minor revisions. However, this unfavorable setting offered the opportunity to compare our findings with those of a traditional V&V process as performed by NASA (mostly inspection); the results of this comparison will be described in the next section.

### 7.4.1 Instantiation of the process

1. Defining a style, structure and language. The NCS specification exhibited a consistent style and structure (conforming to DOD-STD-2167A), and was of overall good structural quality [37]. The language used in the detailed descriptions of each function was concerned mainly with (fairly complex) temporal ordering of input and output events, but also included user interface and other technical issues[2] that influenced the kind of language used. On the other hand, the narrative text was much more elaborate from a linguistic point of view, but since it was intended merely as an explanation of the technical text in the engineering part (that served as the definitive reference), it added no information on its own, and thus was not relevant to our analysis.

2. Selecting desirable properties to check. Given that most of the requirements dealt with assigning certain values to specific output lines upon the occurrence of some event, we decided to perform "black-box" validation of the requirements. In particular, we were interested in the possible values that each output line could assume. Some of the properties we selected at this stage (those that uncovered problems in the specification) are presented later. Formally, input and output lines were described in terms of associated data items (i.e., `/INF`-tagged terms, according to Section 6.2), whose value could change outside system control (for input lines) and whose assignment caused side effects (for output lines).

3. Defining models for checking selected properties. All the properties we defined could be checked on the four models described below:

   - the type (type of data item) model, distinguishing constant values from internal variables and I/O items. Formally, $\forall d \in \text{DataItems}, \mathsf{type}(d)$ is either CONST (a constant value), FLAG (an internal variable) or IO (an input/output line).
   - the defval (default values) model, showing only the default or initialization value of data items, as declared in the requirements. Formally,

[2] As an extreme example, some of the functions described had to behave differently depending on the orbital position of the space station, as expressed by an orbit diagram included in the text. It is typical of lightweight formal methods (and thus of partial validation) that none of the models really need to "understand" the details behind such a diagram, and in our case study we could simply treat a change in orbital position as an unexplained event.

$\forall d \in$ DataItems, $\mathsf{defval}(d)$ is either UNDEF (no initialization value was specified by the requirements) or a set of specific literal values.[3]

- the valspace (value space) model, collecting all the assignments described in the requirements to determine the space of all he possible values for a data item. Formally, $\forall d \in$ DataItems, $\mathsf{valspace}(d)$ is the set of all values whose assignment to $d$ or whose comparison with the value of $d$ is mentioned in the requirements.
- the ecatab (event-condition-action table) model, collecting all the possible actions of the system, together with the conditions and events that cause their execution. Formally, $\forall r \in$ Requirements, $\mathsf{ecatab}(r) = \{\langle events, conditions, actions\rangle\}$ where *events* and *conditions* are predicates on the value of members of DataItems, while *actions* is a set of actions (either assignments to members of DataItems or the special actions "Acquire $d$", signifying the assignment to $d$ of a value read from the hardware, and "Reject $d$", indicating the rejection of a command according to the bus protocol, with $d \in$ DataItems, as specified by requirement $r$.

In Circe's term, definition of a new style and language was accomplished by writing a set of MAS rules specifically targeted at the language used in the NCS specification, and adding some terms to a system glossary. Building this rule set required less than two days of work. Given the highly specialized language used in the document, 30 rules (in addition to the generic ones already described in Chapter 6) were sufficient to obtain complete parsing of three different revisions of the document.

In the same line, definition of new models involved writing modelers to synthesize them from information available in the semantic repository. The implementation of these new modelers, building on the models already included in Circe's library, required approximately 100 lines of AWK code and one day of work. Validation modelers, each only a few lines of code long, were also developed.

Given the MAS rules, glossaries and modelers developed (once only) in the previous step, Circe provided the infrastructure needed to support the production phase of our process.

4. Pre-processing the requirements document. The text of the requirements was simply copied and pasted from the original Word document into our *Source requirements* view, and needed very little manual preprocessing (e.g., commenting out section titles and changing enumerated lists to dash-bulleted lists).

[3]No initializations with non-literal values were specified in our requirements, but if present, they could have been treated as an assignment to be performed unconditionally upon a "Boot" event in the ecatab model. Notice also that although $\mathsf{defval}(d)$ is specified as a set, a double initialization with different values would have been regarded as an error in the requirements ($\forall d \in DataItems, \#\mathsf{defval}(d) = 1$ was one of the desirable properties). In our case study, this property was never violated, and $\mathsf{defval}(d)$ always resulted in a singleton.

Such preprocessing could have been performed automatically if the document size had required it.

5. Parsing of the NL text. The parsing technique adopted by Cico requires that a glossary be defined containing domain-specific terms. This task was accomplished by populating the glossary with:

   - the names of the various data items from the input/output tables included in the specification document,
   - the name of the system itself ("NCS"), and
   - a few other names that were used in the requirements (even though they were not declared as input or output data or command names).

   The parsing process in itself provided a language validation of the requirements. No spelling or syntactic errors were found, supporting our assumption that the requirements were of good syntactic quality [70] with respect to the language defined by our parsing rules. Obviously, only a partial assessment of the semantic quality defined in the same work was performed, by formally validating the models as described below.

6. Building models. The task of building the four models defined in step 6 above was carried out by our newly implemented modelers. Thanks to the modular architecture of Circe, the new modelers could cooperate with pre-existing modelers in Circe's library, thus achieving a large reuse factor.

7. Checking that the models satisfy chosen properties. As said above, checking was carried out by validation modelers. Many of the chosen properties, mostly concerning "obvious" completeness and consistency criteria, were never violated, and are not reported here. Violations that led to the identification of real problems in the specification are discussed in some detail in the following.

### 7.4.2 Findings of the validation

As mentioned above, the valspace model collected all the values mentioned in the requirements as assignable to each data item, either as default values or by explicit statements. One of the properties we wanted to verify on this collection was simply that every non-constant data item had more than a single possible value, or

$$\forall d \in \text{DataItems}, \mathsf{type}(d) \neq CONST \Rightarrow \#\mathsf{valspace}(d) \geq 2$$

We found six different data items (listed in Table 7.1) that did not satisfy this simple property. Closer inspection triggered by this finding revealed that several data items whose labels started with "ACS N1-2" were synonyms to other data items whose names started with "ACS N1S2" — with the N1-2 label (probably) left

| Data item name ($d$) | valspace($d$) | Reason |
|---|---|---|
| "ACS N1-2 Cabin Pressure Lower Limit Warning State" | {TRUE} | Synonym with "ACS N1S2 Cabin Pressure Lower Limit Warning State". |
| "ACS N1S2 Cabin Pressure Upper Limit Warning State" | {FALSE} | Synonym with "ACS N1-2 Cabin Pressure Lower Limit Warning State". |
| "High pressure warning level alarm" | {"return to normal"} | Alarms issued, not set; also synonym with the "Upper limit" alarm. |
| "Low pressure warning level alarm" | {"return to normal"} | Alarms issued, not set; also synonym with the "Lower limit" alarm. |
| "Upper limit warning level alarm" | {"return to normal"} | Alarms issued, not set; also synonym with the "High pressure" alarm. |
| "Lower limit warning level alarm" | {"return to normal"} | Alarms issued, not set; also synonym with the "Low pressure" alarm. |

Table 7.1: Data items failing valspace validation.

over from previous releases. We had originally interpreted these as distinct data items, as the SRS document contained many other distinct data items with only slight variations in their names.

We also had to revise our understanding of the alarm handling by the system. The document used the wording *issue an alarm* to indicate entrance into an alarm state, and *set alarm to "return to normal"* to indicate exiting from an alarm state. We had originally taken these as unrelated operations. Inspection of the relevant section of the SRS document confirmed that issuing an alarm should be interpreted as setting an alarm to "in alarm" — a change in interpretation that was reflected by a simple update of the parsing rule for *issue* in our rule set.

Another issue related to alarms was that the SRS referred to the same alarm in different ways. For example (letters refers to requirements in the SRS):

d. (The NCS shall) *issue a warning level alarm (message: "Node 1 Cabin Pressure Lower Limit Warning Violation")*

e. (The NCS shall) *set the lower limit warning level alarm to "return to normal"*

j. (The NCS shall) *set the low pressure warning level alarm to "return to normal"*

There was potential confusion in the first requirement (d) between the identity of an alarm (a system design issue) and the associated warning message (a user interface issue). Again, inspecting the document with this finding in mind confirmed that the three different designations above were indeed referring to the same entity. The same problem also occurred in three other requirements, and correcting it reduced the total number of alarms from six to two. Tabulation of alarms, as for other I/O items, may have avoided such confusion.

Once we resolved issues in the valspace model, the defval model became amenable to further analysis. So, we tested the model for the property:

$$\forall d \in \text{DataItems}, \mathsf{defval}(d) \neq UNDEF$$

which states that each data item should have a declared default value. Four items did not satisfy this property: the two alarms discussed above, the reported Cabin Pressure value, and the Confirmation Command Rejection Indicator (a flag of the command bus protocol). We already knew that the default state for all alarms was "return to normal" (i.e., no alarm), but the other two data items could potentially provide false information if read before the first assignment.

A better understanding of these potentially dangerous conditions was gained by looking at the ecatab model. This model, inspired by the Mode Transition tables of SCR [55], shows which actions (A) the system performs when a certain event (E) occurs, and which conditions (C) must hold for the actions to be performed.[4] Since the Cabin Pressure Monitoring function was not described in terms of states, we assumed the system to always be in a "Normal" state. The ecatab model synthesized by our tool from the NL text of the requirements is shown in Table 7.2, where we substitute abbreviations for the very long data and command names used in the SRS document. Table 7.2 uses an SCR-like notation: for each requirement $r$, and for each triple in ecatab($r$), conjuncts in the events predicate are marked with "@T" (read: "becomes true"), conjuncts in the conditions predicate are marked with "T" (read: "is true") or "F" (read: "is false") if they are negated, and elements of the actions set are explicitly listed under the Actions heading.

This table passes the usual consistency checks; e.g., disjointness and coverage, under the customary one-input assumption.[5] Given a set of actions $A$, we define its write-set, $W(A)$, as the set of data items modified by the actions in the set, i.e.,

$$W(A) = \{v \mid v := value \in A\} \cup \{v \mid \text{Acquire } v \in A\}$$

Using Table 7.2, we identified the conditions under which the two variables above, Cabin Pressure (P) and Confirmation Command Rejection Indicator (CCREJ), are

[4] The Event-Action model of Section 6.3.3 is a reduced, but more general, version of ecatab.

[5] This assumption holds for events triggered by the reception of commands, due to the serial nature of the 1553 bus used to carry them. We do not know if the one-input assumption actually holds between bus events (command reception) and timer events (sampling cycle, testing read pressure values). In a complete study, the assumption should be verified by inspection of the actual code or detailed design document for our subsystem. The valspace model shows that FDIR and CCREQ can be either ENABLED or DISABLED only, so that checking for either value is sufficient. The system exhibits a certain hysteresis, simply maintaining the previous state for boundary cases where Pressure exactly equals the Upper or Lower Limit.

| Req | Sample Cycle (1Hz) | FDIR = Enabled | P<LL x3 s.c. | P>LL x3 s.c. | P>UL x3 s.c. | P<UL x3 s.c. | D.FDIR | D.FDIR CC | CCREQ = Enabled | E.FDIR | Actions |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a | @T | | | | | | | | | | Acquire P |
| d | | T | @T | | | | | | | | LLWS:=TRUE<br>LLW:=IN_AL |
| e | | | | @T | | | | | | | LLWS:=FALSE<br>LLW:=RTN |
| f | | T | | | @T | | | | | | ULWS:=TRUE<br>ULW:=IN_AL |
| g | | | | | | @T | | | | | ULWS:=FALSE<br>ULW:=RTN |
| h | | | | | | | @T | | | | CCREQ:=Enabled<br>CCREJ:=FALSE |
| i1 | | | | | | | | @T | T | | FDIR:=Disabled<br>CCREQ:=Disabled |
| i2 | | | | | | | | @T | F | | Reject D.FDIR CC<br>CCREJ:=TRUE |
| j | | | | | | | | | | @T | CCREQ:=Disabled<br>FDIR:=Enabled<br>LLWS:=FALSE<br>ULWS:=FALSE<br>LLW:=RTN<br>ULW:=RTN |

Table 7.2: `ecatab` model for the NCS Cabin Pressure Monitoring system.

not initialized (by simply inspecting the rows $r_i$ that satisfy P $\in W(actions(r_i))$ and CCREJ $\in W(actions(r_i))$, respectively). P only occurs in the row for requirement a, so P is not initialized until the first sampling cycle occurs. Depending on the actual subsequent implementation, a first sample could be taken immediately upon startup before even listening to the bus, or it could be delayed for as much as a second — during which time other components of the system reading the pressure value could obtain erroneous (random) results. CCREJ on the other hand is modified only by actions in rows h and i2, so it is not initialized until the first Disable FDIR or the Disable FDIR Confirmation Command (issued while CCREQ is DISABLED, which is its default value) comes up the bus. Since CCREJ is part of the bus protocol, we have to assume that this behavior is documented and does not (currently) constitute a problem.

If the one-input assumption cannot be guaranteed, we can still identify potentially dangerous conditions by examining incompatible requirements. Two action sets $A$ and $B$ are *incompatible* if they assign different values to the same variable, i.e.,

$$\begin{aligned}\mathsf{incompatible}(A,B) \quad \stackrel{def}{\Longleftrightarrow} \quad & \exists v \in W(A) \cap W(B) \text{ s.t.} \\ & (v := a \in A \wedge v := b \in B \wedge a \neq b) \vee \\ & (\text{Acquire } v \in A \cup B)\end{aligned}$$

(for our purposes, "Acquire $v$" assigns to $v$ an unknown value, potentially incompatible with any other value). In order to avoid conflicting assignments to the same data item, the conjunction of the events and conditions of rows with incompatible action sets must always be false, i.e.,

$$\begin{gathered}\forall r_1, r_2 \in \text{Requirements}, \\ \forall \langle evt_1, cond_1, act_1 \rangle \in \mathsf{ecatab}(r_1), \\ \forall \langle evt_2, cond_2, act_2 \rangle \in \mathsf{ecatab}(r_2), \\ \mathsf{incompatible}(act_1, act_2) \Rightarrow \neg(cond_1 \wedge cond_2 \wedge evt_1 \wedge evt_2)\end{gathered}$$

In Table 7.2, the following rows are incompatible: (d,e), (d,j), (f,g), (f,j), (h,i1), (h,i2), (h,j), (i1,j), but only the first and third pair satisfy the above property. In fact, they can be discounted by simple arithmetic (for example, (d,e) represents the case when the Pressure becomes at the same time higher and lower than the admissible Lower Limit). The second and fourth cases are worth flagging as potentially problematic. They represent the case when enabling the FDIR feature continuously (i.e., enabling the reporting of alarms) can actually prevent the alarms from firing, depending on the order in which events are checked by the code. This problem can be avoided by checking that FDIR = Enabled is FALSE as a condition for performing the actions in requirement j. The remaining cases formalize bus protocol violations, and can be ignored for a serial bus.

## 7.5 Lessons learned

The original aim of this study was not the validation of the requirements specification described, but to experiment with the use of CIRCE as a lightweight formal method in an RE process based on NL requirements and inspections. The total effort spent on initial setup, done once only by the author – familiar with the parsing and modeling environment but completely ignorant of the problem domain[6], was less than three working days. Subsequently, parsing the sample specification took less than 10 seconds, while generating and validating the various models took between 0.5 and 2 seconds on a desktop PC. For very large documents, differential parsing and model updating can be used instead of re-parsing the entire document each time. In this case, parsing time drops to 1-2 seconds on average, and is influenced only by the amount of changes in the document between validation iterations, and not by the document size. In our experience, the amount of changes between revisions tends to remain constant, and so is parsing time. Moreover, no training was required for the requirements writers (we used the original document verbatim), and little explanation of how to interpret the results of the process is needed by a V&V team.

Such low computational and human costs strongly suggest that CIRCE could be used successfully during requirements writing and evolution. We believe that the use of a RE supporting environment could also help in tracking the quality of a specification between full releases and inspections, thus providing finer grain monitoring of the RE process and to some extent validating changes that are requested and implemented.

While the inconsistencies we identified in our analysis currently appear to be non-critical, unfortunate experiences (such as the Ariane 5 disaster [91]) have shown us that risk assessments can change as requirements and other circumstances change.

It is particularly interesting to notice that a number of the problems we discovered (e.g., the need for alarms tabulation, the issue/set terminological problem, the need for unequivocal alarm designations and the separation between design and user interface issues) were also independently discovered and corrected by NASA's IV&V team. In fact, the revision of the NCS specification immediately following the three revisions we used in our case study, was changed in line with our findings. The latest revision also included some evolutionary changes that were not prompted by errors in the previous releases, but still presented the errors related to uninitialized and to inconsistently named data items that we discussed above. In light of the partialness of the validation performed by CIRCE, we regard the fact that no error was discovered by the IV&V team that had gone undetected by our study as a mere casual — albeit comforting — occurrence.

Although we examined a single requirements document, the rules and modelers we developed for the case study can be reused to validate subsequent releases of the same document, and indeed can be applied to other projects making use of a similar

[6]This can be an advantage in trying to identify "obvious" errors [10].

document style. So, for example, the effort spent defining parsing rules need not be duplicated in subsequent projects by the same organization.

We believe that the application of our process to an industrial case study served to demonstrate the feasibility and benefits of lightweight formal methods in this context, and the effectiveness of CIRCE as a lightweight validation tool. The low cost of our approach, both in terms of human training and of computational resources needed, makes it particularly well suited for the initial introduction of formal methods in an organization.

# Chapter 8

# The Light Control System

Abstract

In this chapter a second case study is presented. We used CIRCE for the analysis and refinement of an informal customer's description of a system to control the light system of a building. The focus of the case study was on writing precise, correct and detailed requirements for a light control system starting from the customer's descriptions of the problem domain and of his needs regarding the system.

The support offered by CIRCE allowed us to identify a number of problems in the original customer's description, that were almost invariably confirmed by the customer. It also ensured that the requirements we wrote did not present similar errors. Inspection of several models in their graphical form was used, together with the indications of validation modelers, to perform continuous validation of the requirements during their writing.

## 8.1 The case study

The subject of the case study was a Light Control System for a building. The system featured extensive environment control, human interaction, and handling of exceptional cases, thus covering a number of typical issues in embedded control systems.

The Light Control System (LCS) was originally introduced in [41], and a later version was used as a common case study in a recent Dagstuhl Seminar [17]. Besides having been used as a testbed for comparing requirements elicitation and specification techniques, the case was appealing because it concerned an already implemented system, and thus assumptions made were easily verifiable on the field.

The problem description document (PDD), 13 pages long, contained the *user requirements* ("informal needs") for the LCS. This was an important difference with respect to the NCS case study in Chapter 7, in which the source document contained detailed *engineering requirements*. In the LCS case, user requirements were often vague and ambiguous (e.g., "no hazard for persons is allowed"), and in some case contradictory requirements were stated. Our goal was thus not to validate the PDD, but rather to incrementally translate it into a precise and consistent NL requirement document, using CIRCE to support the translation and validate the resulting document.

The PDD was divided in four parts. The first one detailed the motivations for the project and introduced typographical conventions used in the rest of the document. The second part described the physical layout and the available infrastructures for the 4$^{th}$ floor of Building 32 of the University of Kaiserslautern campus, where the LCS was located. Part 3 listed informal needs for the LCS, separating functional needs and non-functional needs. Functional needs included user needs and facility manager needs. Non-functional needs included fault tolerance, safety and legal aspects, and user interface issues. A dictionary of terms of the application domain, in Part 4, concluded the document.

## 8.2 Translating and validating the PDD

The first step in our translation was to populate the glossary with terms from Part 4 of the PDD. Almost no tags were assigned in this phase, except for a few terms (e.g., "control system", "control panel", etc.) that were assigned the `/SYS` tag to mark them as parts of the system we were asked to state requirements for.

The second step consisted in further describing the domain by translating — often almost literally — statements from the second part of the document in terms of known or newly-introduced relations. For example, the original statement in the PDD

> "The fourth floor of Building 32 consists of three sections and shares two staircases, staircase east (SCE) and staircase west (SCW), with other floors of the building, as shown in Figure 8.1. Each section is divided into some hallway sections (H) and rooms, each of which may be an office (O), a computer lab (CL), a hardware lab (HL), a peripheral room (P), or a meeting room (M). All rooms in a section are accessible via a connected hallway section. There are three hallway sections and 22 rooms to control."

was translated into

1. The fourth floor of Building 32 consists of three sections and two staircases.

2. Each section is divided into some hallway sections and a number of rooms.

3. Offices, computer labs, hardware labs, peripheral rooms and meeting rooms are rooms.

4. All rooms in a section are accessible via a hallway section.
5. The system controls three hallway sections and 22 rooms.

Part of the original description was omitted in our version. The schematic drawings illustrating the architecture of the floor (Figure 8.1), and the tables detailing geometric, physical and electrical characteristics of the analog sensors used by the systems were not translated into requirements. Most of the information contained in the omitted parts was indeed important for actual implementation (e.g., for writing hardware drivers for the sensors), but not relevant to the requirement phase. Abbreviations (e.g., CL for "computer lab") were simply introduced as synonyms in the glossary.

In performing this step, we introduced a few more terms in the glossary (e.g., the *is_accessible_via* relation) that were not included in the document dictionary. Missing terms were immediately identified by CIRCE's document-related views, in particular through parsing rests and abstraction finding. In a few cases when the correct tagging was not immediately apparent, speculative tagging was used to determine the correct set of tags.

The 23 statements resulting from this activity constituted a reasonably accurate description of the domain in which the LCS operate. The models built on this description could be inspected in graphical form both to verify our understanding, and to check for possible incompleteness in the PDD. A few stylistic problems were already discovered in this phase: for example, a number of properties were predicated individually on each kind of rooms, while others were generically referred to the entity "room". The *Subsumption* and *Approximate subsumption* modelers (Section 6.1.4) identified such cases, and also identified an asymmetry between Peripheral rooms and all other kinds of rooms. In fact, the PDD described the rooms in five distinct sections, one for each kind of room. Offices were described in detail:

- **Offices.** Each office has one door to the hallway section and can have up to two doors leading to its adjacent rooms. Each door is equipped with a door closed contact.

  Each office is equipped with:

  - one motion detector [...];
  - two ceiling light groups [...];
  - [...]
  - two status lines [...].

Other rooms, however, were only described by difference from the offices,[1] i.e.,

[1] CIRCE does not currently offer any language construct to express this type of "copy with modify" semantics, thus in the translation the similarity was expressed by collecting relevant rooms in a single statement, e.g. "Each office, computer lab, hardware lab and meeting rooms

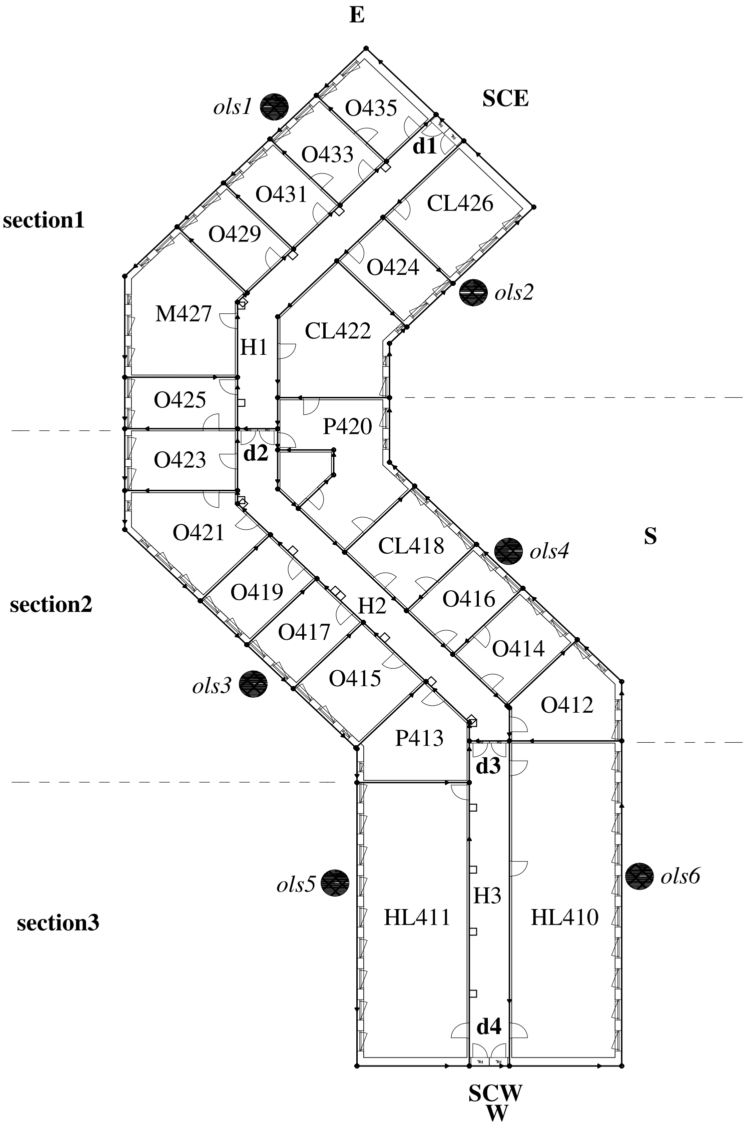



Figure 8.1: Architecture of the 4th floor of Building 32 at Kaiserslautern University (picture taken from [15]).

- **Computer lab.** Same as office.
- **Hardware lab.** Same as office, but with more than one door leading to the hallway section.
- **Meeting room.** Same as office.
- **Peripheral room.** The peripheral rooms will not be controlled by the control system, and thus they are not described here.

In the approximate subsumption model, peripheral rooms were singled out for not participating in any relationship (and not having any attribute) shared by the other rooms. A better description would have been obtained by distinguishing *controlled rooms* (offices, labs and meeting rooms) and *other rooms* (peripheral rooms). The subsumption model would then suggest lifting all the relationships and attributes common to all *controlled rooms* (i.e., everything except for the number of doors leading from a room to the hallway) to the *common rooms* class, and some of these to the general *room* class.

It is interesting to notice that the simpler classification hierarchy for rooms established by the customer in the PDD led to serious interpretation problems in other parts of the document. For example, at some point a distinction was made between "offices" and "remaining rooms". A strict interpretation would have read "remaining rooms" as "all the rooms that are not offices", i.e. computer labs, hardware labs, meeting rooms, and peripheral rooms. It was not immediately clear from the context that indeed "remaining *controlled* rooms" was intended, and thus that the properties described did not apply to peripheral rooms. The more precise hierarchy suggested by CIRCE would have avoided this kind of ambiguity in the first place.

Other minor issues in the PDD (and in our translation of the PDD) were also identified by the selection modelers (Section 6.1.5). For example, a possible ambiguity was discovered in the expression "the motion detector of a door", that admitted two different interpretations:

- the motion detector placed above a door, or
- the motion detector that detects motion in proximity of a door.

Common sense and some domain knowledge[2] would let us conclude rapidly that both interpretations select the same motion detector, but CIRCE's extreme ignorance of the ways of the world makes it a stricter analyst [10]. In fact, examining the rationale provided by the tool for its interpretation, we are reminded that each door has *two* motion detectors, one on each side, and thus the expression "the motion detector of

is equipped with ...". Notice that the "same as ... but ..." linguistic device is *not* expressing inheritance, for which CIRCE has appropriate constructs.

[2]In this case, knowing the fact that motion detectors can only detect motion in proximity of the place where they are installed.

a door", in itself, is seriously ambiguous. We need further inference (e.g., the fact that the system controls lights in hallways but not in staircases) or further context (e.g., the fact that a single hallway section is being considered) to interpret such a statement correctly.

In one case, the selection modeler identified a missing relation between "status line" and "hallway", whereas a statement

- The status line of a hallway shows the status of the ceiling light group of the hallway section.

was used. Indeed, in the translation process we had erroneously omitted part of a statement establishing an *is_equipped_with* relation between hallways and status lines. Similar occurrences show the importance of checking properties on the product of the requirement process as a means to remedy to errors (even clerical, as in our case) in the requirement writing process.

Functional needs from the third part of the PDD were translated into requirements as for the domain description statements in the second part. However, while domain description statements were mapped almost exclusively to declarations of relationships and attributes, functional needs were mapped largely on functional- and behaviour-oriented requirements.

Language frames recognized by Circe do not allow the expression of patently incomplete requirements. To obtain correct parsing of the requirements, in several cases we had to complete the literal translation of the PDD with more explicit information. For example, the Facility Manager need

**FM7** If a malfunction occurs, the facility manager has to be informed.

was translated as

**FM7** If a malfunction occurs, the system sends a malfunction warning to the facility manager control panel.

Notice how the passive voice sentence "the facility manager has to be informed" was translated into its more detailed, active voice version above. This is an instance of the *completion* stereotypical operation on requirements [63]: information is added to fill the knowledge slots in a given template. In our case, we had to add three distinct facts to the original requirement:

1. the fact that the system is the initiator of the action (and thus responsible for verifying the condition),

2. the fact that interaction with the facility manager is through the facility manager control panel, and

3. the fact that the facility manager is "informed" through a warning message.

Another example of translation with completion comes from user need U5:

**U5** The push buttons for the window- and wall-ceiling light group in a room should show the following behaviour:

(i) if the corresponding ceiling light group is completely on, it will be switched off;

(ii) otherwise it will be switched on completely.

that is translated as

**U5** If the push button of a room is pushed,

(i) if the light group corresponding to the push button is completely on, the system sends the "switch off" command to the light controller of the room;

(ii) otherwise the system sends the "switch completely on" command to the light controller of the room.

Another stereotypical operation on requirements that was used throughout the translation was *linguistic refinement*. Refinement consists in substituting a high-level atomic term with a lower-level, structured expression. For example, in FM7 above, "a malfunction occurs" is defined as an atomic event (`/EVT`, Section 6.3.3) in the glossary. Through refinement, the same event can also be described as

When a component is malfunctioning ...

where "malfunctioning" is a possible state (`/MODE`, Section 6.3.4) for every "component" of the system. Notice that in refining the original event we have introduced more information that is analyzable by CIRCE, in this case that our system is made of components and that components can be malfunctioning. This results in more precise definition of our requirements. For example it is now clear that the expression "a malfunction occurs" only concerned malfunctions of system components, that are explicitly listed in the requirements, and not malfunctions of other entities in the domain.

Moreover, more complete validation is possible on refined requirements. In our example, the *BadAutomata* modeler of Section 6.3.4 would warn that "components" should have more than a single state, thus prompting us to write a full declaration stating that components can be either malfunctioning or working correctly. After this declaration, the same modeler would point out that it is not described anywhere in the requirements what happens when a malfunctioning component is repaired, and begins working correctly. In fact, this issue is never mentioned in the PDD, although a competent designer could figure a reasonable implementation (e.g., periodically polling malfunctioning components to check whether they have been repaired).

Systematic treatment of requirements as shown above has the added benefit that hypothesis made during the translation (in particular, in refinement and completion) can be documented and checked with the customer or other interested parties.

Translation of the functional needs in the PDD resulted in 30 requirements of varying complexity (with parse trees containing up to 15 nodes). To these, translations of non-functional needs were added.

Some of the non-functional needs could not be expressed in terms of models known to CIRCE (e.g., NF8: "The control panels are easy and intuitive to use"), and were simply inserted (although not parsed correctly) for reference. The remaining needs were translated as above, often in functional terms. For example,

**NF2** If any outdoor light sensor does not work correctly, the default light scene for all rooms is that all ceiling light groups are on.

was translated as

**NF2** If the outdoor light sensor of a room is malfunctioning, the system resets the default light scene of the room to the emergency light scene.

Notice how in this translation we were induced to complete the requirement by transforming the impersonal "the default light scene ... is that all ceiling light groups are on" to an explicit action performed by the system. Notice also how we introduced an explicit "emergency light scene" corresponding to (but conveying more information than) setting all light groups to on.

Our version of requirement NF2 also introduces some correction originated from verification of our understanding with the customer. In fact, the original form of the requirement specifies that *all* rooms should have their lights on if *any* outdoor light sensor fails. This is excessive, and conflicts with other non-functional requirements asking to minimize electrical power consumption. However, it is formally consistent, and thus no amount of formal validation could flag it as an error.

The customer's real intention was that the system would turn on the lights in all those rooms for which the calculation of available daylight depended on the malfunctioning outdoor light sensor (there were six sensors, corresponding to different orientations of the rooms' windows, as shown in Figure 8.1). Hence, we used the selection expression "of a room" both for the failing sensor and for the light scenes. After this correction, however, CIRCE's validation modelers signaled the missing relationship between outdoor light sensors and rooms (a relationship that was implicit in Figure 8.1, but not stated anywhere in the requirements). Explicitly declaring the "same orientation" relation solved the problem, making requirements more precise once more.

Finally, the fourth part of the PDD, the Dictionary, was translated. Of the 40 terms comprising the dictionary, 37 were already in CIRCE's glossary. The remaining three terms were either not needed in the requirements, e.g., "architecture: structure of a building, floor or room", or falsely atomic, e.g., "hallway sections ceiling light group". In place of the latter term, we had used the selective expression "ceiling light group of a hallway section". We had also used the corresponding "ceiling light group of a room" where needed, while the PDD did not contain the symmetric term "rooms ceiling light group".

A few definition called for explicit requirements, mainly to establish appropriate classification hierarchies. For example, from the definition:

> Motion detector: *sensor detecting motion of a person or animal in its range.*[3]

we extracted the domain description statement:

> Motion detectors are sensors.

(we could also extract the fact that motion detectors have an attribute "range", but this attribute was never needed in the requirements document).

At the end of the translation process, our requirements document consisted of 62 statements, of which 28 were domain description statements, and 34 expressed functional and non-functional requirements.

The entire translation step required three days of work, including the time needed to familiarize with the domain. No source of information outside the PDD was taken into account, and no customer interaction was available during that time. We believe that effort could have been substantially reduced had the customer been available for comments during the translation process.

## 8.3 Validation findings and customer feedback

As discussed above, the process of translating statements from the PDD into requirements accepted by CIRCE uncovered a certain number of defects in the original document. These defects were in part identified by the validation modelers in CIRCE's library, and in part by the effort of rewriting statements to fit CIRCE's linguistic templates.

Every assumption made in resolving the issues in the PDD was recorded for subsequent verification with the customer. In some case, later assumptions dominated previous ones, and the former were deleted. Only problems that hindered the faithful translation of the document were considered at this stage; problems with the system described by the requirements were left for a later step. Overall, slightly less than 50 assumptions and open questions were left at the end of the translation phase. Of these, 16 were selected as high-priority issues, i.e., issues that could not be

[3] This definition is obviously approximate: a motion detector would not distinguish a moving person or animal from anything else, say a moving robot. This however is typical of customers' problem descriptions, and mostly irrelevant for our purposes.

resolved without further information from the user, or cases in which one of several possible solutions was opted for in taking some assumption.

The 16 most relevant questions and assumptions were then sent to the customer for answer and verification. The customer replied to only 12 questions, confirming the assumptions we made in eight of them. In the remaining four cases, information missing from the PDD was provided (e.g., we discovered that each door had a small window above it through which light could fall into the hallways, whereas Figure 8.1 suggested that hallways were not exposed to daylight).

The four questions to which the customer did not reply were mainly concerned with the structure of the document (rearranging sections, merging requirements) and with definitions in the dictionary. Although no comment was obtained on these four questions, a subsequent version of the same document was arranged as suggested by three of them, while the fourth question was no longer applicable (a term whose definition was questioned had been removed from the dictionary).

Customer interaction convinced us that most of the problems we had discovered in the process of translating user needs into requirements, and then through the validation modelers we had available, were real defects in the problem description document. Preliminary results seem to indicate that a significant portion of those problems were also independently identified by other researchers working on the same case study. Unfortunately, precise comparative data were not available to conduct a formal assessment of the effectiveness of our technique with respect to other approaches.

## 8.4 Subsequent activities

Our goal in this case study was to test the effectiveness of Circe in analyzing and refining customers' documents describing a problem and its domain. It was not our intent to do a complete requirement analysis on the Light Control System. However, in this section we point out some of the work that was left for later stages, by using Circe's views to establish a roadmap of activities.

Figure 8.2, for example, shows a simplified entity-relationship view of the system described by the (translated) requirement document. In that figure, labels for relations and attributes have been hidden to avoid excessive clutter, and some basic relationship (i.e., is-a and part-of) have been filtered out for the same reason.

The automatic layout algorithm used by the graph translator has placed a number of entity nodes in the same position (the stack of nodes that has "computer lab" on top). It can be easily observed that all those nodes, one for each kind of room, share the same relationships with other nodes in the diagram. Indeed, this observation corresponds to the approximate subsumption modeler suggestion that such relationships could be moved to the parent class "room", if not for the "peripheral room" that can be observed, isolated, on the right hand side of the picture.

One of the first actions that should be taken in further refining the requirement

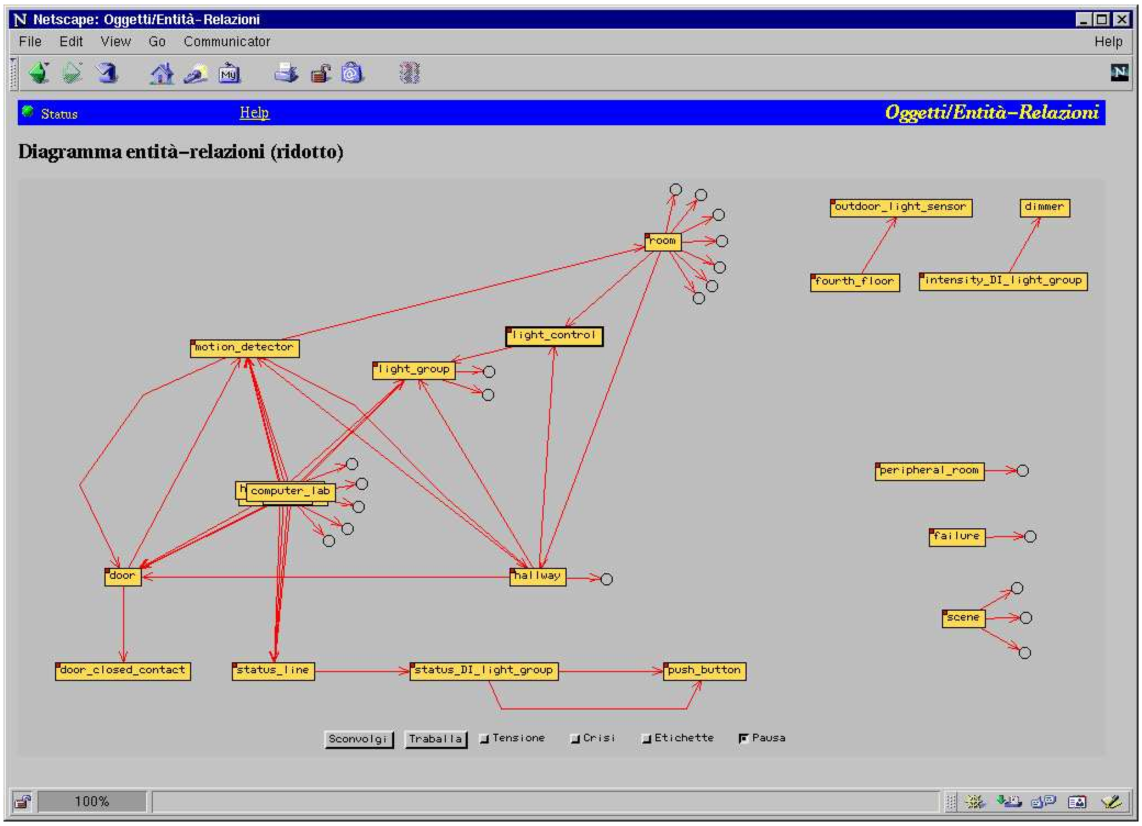



Figure 8.2: E-R view for the translated PDD.

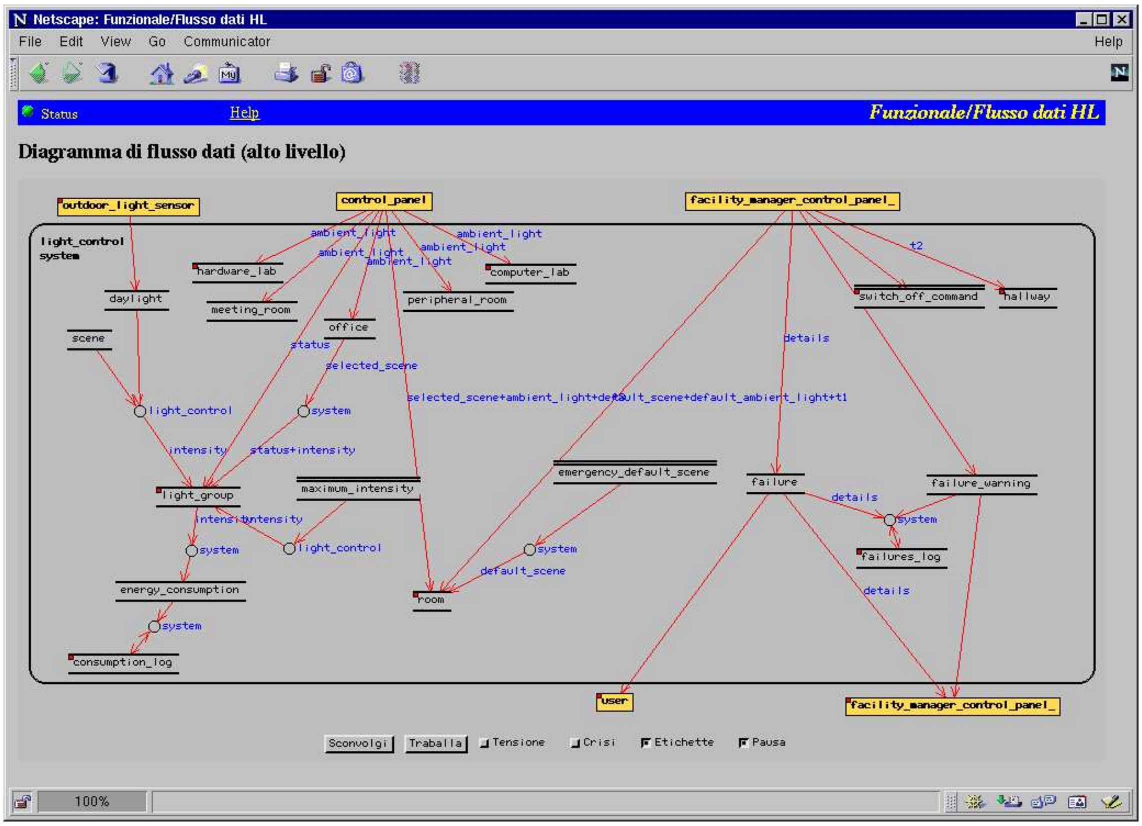



Figure 8.3: High level data flow view for the translated PDD.

document would then be to reorganize the class hierarchy for rooms, either by introducing new intermediate classes as suggested at page 149, or by dropping peripheral rooms from the description of the domain. This refinement alone would greatly simplify the entity model.

The graph in Figure 8.2 is broken in six components, probably too many: an indication that the domain description is not accurate enough. When is-a and part-of relations are shown, however, the graph becomes more strictly connected. Still, the outdoor light sensor entity does not participate in any relationship beyond the simple "is part of the equipment for" with the fourth floor entity. In fact, a relationship expressing the fact that a room has the same exposure as a given outdoor light sensor should be added, as mentioned above.

While domain description could be marginally refined beyond what we have already suggested, the inner workings of the system are only sketched up to this point. This is not surprising, given that our requirements document only contains direct translations of user's needs, and are not detailed enough to guide the design phase.

Figure 8.3 shows the high level data flow diagram for our system. As can be observed, most flows in the system are incomplete, and the only output described explicitly are warnings sent to the user of a room and to the facility manager when some component of the system fails to operate correctly. Most repository are never brought outside the system. In particular, the PDD did not describe any way to access the energy consumption log (on the bottom left corner of the system box in Figure 8.3) and the failures log (on the right, above the facility manager control panel). Moreover, considerable asymmetry can be observed between offices and other rooms. Settings for the light groups are in certain places described as a pair (status,intensity) computed according to the selected scene of an office, and in other cases as a simple intensity value computed from a generic "scene" (taking into account daylight intensity measured from outdoor light sensors).

While some of the unpleasant features in the data flow model may have been introduced by inaccurate translation, most of them are a direct result of ambiguities, inconsistencies, and underspecifications in the PDD.

All these potential problems, and others, should be resolved in order to obtain a requirements document that is precise and detailed enough for the design phase. The possibility to access interactive and dynamically updated diagrammatic representations of models of the system described in the requirements document is in our opinion an invaluable tool to rapidly identify problems[4] and evaluate possible solutions. At the same time, the availability of a textual, natural language description of the system simplifies interaction with possibly non-technical customers and other interested parties, thus simplifying user validation.

## 8.5 Lessons learned

In performing this case study, we have tested Circe's ability to support the activity of writing requirements for a new system even in its initial stages. The mostly literal translation of statements expressing user's needs about the LCS was sufficient to Circe to build an initial model of the new system and of its environment. Validation modelers were used both to identify a number of problems in the original customer's document, and to plan a roadmap for further refinement and completion of a more precise requirements document. Interaction with the customer confirmed that the vast majority of the problems identified in the PDD were indeed real problems needing clarification or further explanation.

The case study also served to indicate opportunities for the extension of Circe's library of modelers towards the handling of non-functional requirements. The extension does not impact linguistic aspects (in fact, rules to parse non functional requirements could be easily added to Cico's rule set), but rather reflects some immaturity in current practices when it comes to building and reasoning on mod-

[4] All the issues we have discussed, with the exception of asymmetries between offices and other rooms, are also identified by validation modelers. This makes it possible to only examine diagrams when problems are identified during the requirements engineer's work, and automatize partial formal validation of a system.

els of non-functional requirements. Extending Circe's capabilities in this direction requires defining new models for certain families of non-functional requirements, together with their associate representations, validation properties and possibly metrics. In this spirit, [4] describes a possible model for structural requirements, i.e., non-functional requirements that have an impact on the architecture of a software system.

While a complete assessment has not been conducted, comparison with other reported experiences [17] seems to indicate that the support offered by Circe to the requirement writing activity scores comparably with other techniques in terms of accuracy, while requiring less expertise and effort on the part of the requirement writer. Moreover, since the final product is a natural language requirements document, user validation can be performed without the need for translating back formal requirements into a language comprehensible to the user (a translation that could inject insidious errors in the process). We believe that this fact can substantially improve interaction between the requirements writer and the customer, and that improved interaction leads to requirements of better overall quality.

# Conclusions

In this thesis we have studied the problem of providing automatic support to a number of important activities in the context of requirements engineering. The main contributions of the thesis are summarised below.

**On the conceptual framework**. We have introduced a conceptual framework for requirements engineering based on linguistic properties of requirements and on the definition of basic activities performed on them. In particular, by separating lexicon, syntax, semantics and pragmatics we have been able to give a rational characterization of such fundamental activities like requirements expression and understanding, and to the explicit or mental processes of building models from the requirements, of validating those models, and of obtaining metric data on them.

**On requirements engineering environments**. The conceptual framework we defined allowed us to state a set of requirements for an environment supporting the activities we have focused on. We exhibited an architecture satisfying those requirements, and presented the CIRCE environment we designed and implemented according to our architecture.

Our architecture is highly modular, scalable, and easily extendible. This is a crucial characteristic for a requirements engineering supporting system that must accommodate different needs in different contexts. Our realization of the architecture does not claim completeness. Rather, each organization should be able to extend the basic set of functionalities provided by CIRCE by adding specific expertise and know-how. Another important direction for customization is in style guides for requirements document and in quality assurance procedures. We have shown in the Space Station's case study how CIRCE can be extended to accommodate a new language, style, domain, models, and validation criteria with a reasonable effort.

**On the use of natural language**. We have shown that natural language has a number of important advantages when it comes to expressing requirements for a software system. First of all, it is the only language that one can confidently assume that is shared among all the participants in a software development process (including the customer, the system users and other potentially non-technology savvy stakeholders). Moreover, it is universal, in the sense that it can predicate any property on any model that can be "read aloud" or that can be explained to anybody not already familiar with its formal meaning.

The well-known shortcomings of NL come mainly from its fuzzy semantics. Most of those problems, however, can be eliminated by exactly defining the terms of reference in the world we are describing (as we have done through our glossary and associate tags), by assigning a precise semantics to a restricted set of natural language (as we have done via our MAS rules and via the interpretations given to the information in the semantic repository by the various modelers), and by assuring that this interpretation is prominently shown back to the participants in appropriate forms (as we have done in our *Parsing report* view, or by graphical rendering of models) for feedback and analysis.

Natural language has also the unique capability of gluing together a large amount of disparate information in a single "knowledge packet". Consider for example the requirement

> If the system is running remotely, the permissions of a user expire after 5 minutes since the system received the last transaction from the user.

Not only this requirement specifies that

1. our system can run either remotely or in some other state (FSA model);
2. a user of the system has an associate (set of) permissions (E-R model);
3. a user can send transactions to our system (DFD model);
4. a data path must exist between the user and our system (Communications model):
5. permissions may expire after 5 minutes of inactivity (behaviour/time);
6. permissions may be expired or in some other state (e.g., "valid") (data properties)

but it also carries "hidden" information, for example that

7. facts 1-6 above constitute a single knowledge unit;
8. design links exist and must be maintained among all the different incarnations of this requirement in different models;
9. this requirement is similar to other requirements on the same subject (Correlation model);
10. we have indeed some use for those "system", "user", "transaction" etc. terms that we have included in our domain (Glossary usage model);
11. our system has at least a certain complexity (MFeP metric);
12. adding or removing the requirement is a single conceptual operation on the requirements document (process metrics).

Despite the progresses made by requirement engineering environments based on formal models, to the best of our knowledge none of the existing proposals can handle both the explicit and the implicit information in a simple requirement as efficiently as just stating it in natural language permits.

On the other hand, NL requirements can lack depth and precision. We believe however that requirements analysis should not aim at absolute precision as its primary goal. The hidden risk with many formal methods is that the requirements writer ends up specifying with the utmost exactness the wrong system. It is our opinion that requirements analysis should concentrate in identifying the right system first, and in ease of change, experimentation, and interaction with other stakeholders, leaving the goal of absolute detail and precision to subsequent specification and design phases.

We believe that the architecture we proposed and its implementation, the array of models that we presented in Chapter 6, and the outcome of our case studies constitute, if not a formal proof, at least convincing evidence that natural language can successfully be used to specify requirements for a software system, and that its shortcomings can be reduced to a comfortable level by rigorous definition of our models and of the properties that we want to hold on them.

**On parsing natural language requirements**. We have described in Chapter 5 a new technique for shallow parsing of natural language requirements that is accurate, computationally efficient, and easy to extend to new domains. Differently from traditional approaches, our technique exploits domain knowledge already during the parsing stage, and can be applied to a combination of grammatical and domain knowledge if needed. Our parsing algorithm includes facilities (e.g., speculative tagging) to support non-technically educated users, in keeping with the general requirements we stated for the environment.

Experiences accumulated on many projects, both in academic and in industrial environment, convince us that domain-based parsing provides a good linguistic coverage of relevant domains. Evidence of good coverage is given in Chapters 7 and 8. In the first case, our parser was able to correctly and completely parse three revisions of a set of requirements not written specifically for automatic processing (but targeted at human inspection, according to NASA's IV&V process). In the second case, the language accepted by the parser proved flexible enough to allow us an almost literal translation of the customer's "needs". In detail, for slightly less than half of the statements a strict word-by-word translation was used, while the remaining statements only needed minor adjustments — always improving the quality of the requirement.

**On models, validation, and metrication**. A wide collection of models has been presented. These models, together with associate validation checks and metric data, constitute the basic set of models supported by CIRCE. Beside the intrinsic value of each model, that is in certain cases relevant, the *integration* among models, and between models and the natural language requirements that introduce them, provides an added value that is not easily found in traditional modeling techniques.

We have shown above how a single requirement can impact a number of different models. The same network value is found in validation: although any single validation check is fairly simple, it is very difficult for a defect in the requirements to escape *all* the different validation checks. Also, a single operation (e.g., adding a requirement to a document) impacts in the same way a number of different metrics. Integration between different models is a key factor for an effective support of requirements engineering activity.

**Open problems and future goals**. A number of problems that we have identified in the course of our study have not been resolved satisfactorily yet. In a thesis whose aim is to provide support to such a rich activity as requirements engineering, there is always room for improvement. Among the problems that are still open we cite:

- Providing input methods based on direct editing of diagrams representing models. In certain cases, direct editing of natural language requirements can be inconvenient, and for skilled users direct manipulation of models through their representations may be more practical.

- Providing better integration with standard modeling languages like UML [14]. A mapping of some of the models we have defined into UML would permit a more precise assessment and refined comparisons of our results with that obtained through other methods. Moreover, given the level of support already available for UML, this integration would favour more direct links between requirements writing and analysis and other activities in the software development process.

- Enriching the basic set of modelers, projectors, and translators with support for more domains. While many domains are too specific to be treated in a general environment, some improvement (e.g., moving from Event-Action tables to full SCR) would be of wide applicability, and could well be part of the basic support offered by CIRCE.

More importantly, we have defined a conceptual and technical framework in which controlled experiments in requirements writing and analysis can be conducted. The results we presented in this work constitute an enabling factor for further research in this direction. Problems like requirements process metrication, requirements document structural analysis, effects of fine-grained process guidance on requirements writing, and others can be attacked using the conceptual and technical tools we developed. We have the hope that this will favour advancement in our discipline.

# Bibliography


[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, 1988.

[2] V. Ambriola, R. Di Meglio, V. Gervasi, and B. Mercurio. Applying a metric framework to the software process: an experiment. In B. C. Warboys, editor, *Proceedings of the Third European Workshop on Software Process Technology*, pages 207–226. Lecture Notes in Computer Science Nr. 772, Springer–Verlag, Feb. 1994.

[3] V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proceedings of the Twelfth IEEE Conference on Automated Software Engineering*, pages 36–45. IEEE Press, Nov. 1997.

[4] V. Ambriola and V. Gervasi. Representing structural requirements in software architecture. In R. Nigel Horspool, editor, *Proc. of the IFIP WG2.4 Conference on Systems Implementation 2000: Languages, Methods and Tools*, pages 114–127. Chapman & Hall, Feb. 1998.

[5] V. Ambriola and V. Gervasi. Experiences with domain-based parsing of natural language requirements. In G. Fliedl and H. C. Mayr, editors, *Proceedings of the Fourth International Conference on Applications of Natural Language to Information Systems*, number 129 in OCG Schriftenreihe (Lecture Notes), pages 145–148, 1999.

[6] V. Ambriola and V. Gervasi. Process metrics for requirements analysis. In *Proceedings of the Seventh European Workshop on Software Process Technology*, Lecture Notes in Computer Science. Springer-Verlag, Feb. 2000. (to be published).

[7] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering*, pages 158–165. IEEE CS Press, May 1991.

[8] R. Balzer, N. Goldman, and D. Wile. Informality in program specifications. *IEEE Transactions on Software Engineering*, 4(2):94–103, Mar. 1978.

[9] L. Bannon. From retrospective to prospective: The next research agenda for CSCW. In *Proceedings of the ACM 1996 Conference on Computer Supported Work*, pages 440–440, New York, Nov.16–20 1996. ACM Press.

[10] D. M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, Feb. 1995.

[11] B. W. Boehm. Software engineering: R & D trends and defense needs. In P. Wegner, editor, *Research Directions in Software Technology*, page 717. MIT Press, Cambridge, MA., 1979.

[12] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, Jan. 1984.

[13] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique* LOTOS, pages 23–73. Elsevier Science Publishers, 1989.

[14] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language, version 1.0*. Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, 1997.

[15] Journal of Universal Computer Science. Special issue on “Requirement Engineering: The light control case study”. Problem Description, Sept. 1999. (Published as part of the call for paper for the special issue).

[16] E. Börger. Why use of evolving algebras for hardware and software engineering. In M. Brtosek, J. Standek, and J. Wiedermann, editors, *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'95)*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer-Verlag, 1995.

[17] E. Börger, B. Hörger, D. Parnas, and D. Rombach, editors. *Requirements Capture, Documentation, and Validation*, 1999. Dagstuhl Seminar Report 242.

[18] E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997. (Special ASM issue).

[19] P. G. Brown. QFD: Echoing the voice of the customer. *AT&T Technical Journal*, 70(3):18–32, Mar.-Apr. 1991.

[20] F. Brum-Cottan and P. Wall. Using video to re-present the user. *Communications of the ACM*, 38(5), May 1995.

[21] J. Bubenk et al. Facilitating 'fuzzy to formal' requirements modelling. In *Proc. of the 16th IEEE International Conference on Requirements Engineering*, pages 154–157, Los Alamitos, California, Apr. 1994. IEEE CS Press.

[22] C. P. C. Rolland. A natural language approach for requirements engineering. In P. Loucopoulos, editor, *Advanced Information Systems Engineering*, number 593 in LNCS. Springer-Verlag, 1992.

[23] B. L. Charlier and P. Flener. Specifications are necessarily informal, or: Some more myths of formal methods. *Journal of Systems & Software*, Mar. 1998.

[24] R. Cole, J. Mariani, H. Uszkoreit, G. B. Varile, A. Zaenen, A. Zampolli, and V. Zue, editors. *Survey of the State of the Art in Human Language Technology.* Studies in Natural Language Processing. Cambridge University Press, Mar. 1998.

[25] A. Cucchiarelli et al. Supporting user-analyst interaction in functional requirements elicitation. In *Proceedings of the First Asia-Pacific Software Engineering Conference*, pages 114–123, Los Alamitos, California, Dec. 1994. IEEE CS Press.

[26] J. L. Cybulski and K. Reed. The analysis and specification of systems and software requirements. *IEEE Software*, 9(3):62–68, Mar. 1992.

[27] N. D. da Costa. On the theory of inconsistent formal systems. *Notre Dame Journal of Formal Logic*, 15:497–510, 1974.

[28] H. Dalianis. A method for validating a conceptual model by natural language discourse generation. In *Advanced Information Systems Engineering*, number 593 in LNCS. Springer-Verlag, 1992.

[29] A. Davison. Readibility formulas and comprehension. In L. R. Roehler, G. G. Duffy, and J. Mason, editors, *Comprehension Instruction; Perspectives and Suggestions*. Longman, New York and London, 1984.

[30] T. Dierks and C. Allen. The TLS protocol, version 1.0 (RFC 2246). Technical report, The Internet Society, Jan. 1999.

[31] J. M. Drake, W. W. Xie, W. T. Tsai, and I. A. Zaulkernan. Approach and case study of requirement analysis where end users take an active role. In E. Straub, editor, *Proceedings of the 15th International Conference on Software Engineering*, pages 177–186, Baltimore, MY, May 1993. IEEE Computer Society Press.

[32] D. Duffy, C. MacNish, J. McDermid, and P. Morris. A framework for requirements analysis using automated reasoning. *Lecture Notes in Computer Science*, 932, 1995.

[33] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Co-ordinating distributed viewpoints: the anatomy of a consistency check. *Concurrent Engineering: Research and Applications*, 2(3):209–222, 1994.

[34] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirement modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, Jan. 1998.

[35] K. El Emam, S. Quintin, and N. H. Madhavji. User participation in the requirements engineering process: an empirical study. *Requirements Engineering Journal*, 1(1), 1996.

[36] R. G. Epstein. The use of computer ethics scenarios in software engineering education: the case of the killer robot. In J. L. Diaz-Herrera, editor, *Proceedings of the Seventh SEI CSEE Conference*, number 750 in Lecture Notes in Computer Science, San Antonio, USA, 1994. Springer-Verlag.

[37] F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi, and S. Ruggieri. Achieving quality in natural language requirements. In *Proceedings of the Eleventh International Software Quality Week*, San Francisco, USA, May 1998.

[38] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[39] M. S. Feather. Rapid application of lightweight formal methods for consistency analyses. *IEEE Transactions on Software Engineering*, 24(11):949–959, Nov. 1998.

[40] Y. A. Feldman and C. Rich. Principles of knowledge representation and reasoning in the FRAPPE system. In *Proc. of the* $6^{th}$ *Israeli Symposium on Artificial Intelligence*, Dec. 1989.

[41] R. L. Feldmann, J. Münch, S. Queins, S. Vorwieger, and G. Zimmermann. Baselining a domain-specific software development process. Technical Report SFB501 TR-02/99, University of Kaiserslautern, 1999.

[42] N. Fenton and S. L. Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2 edition, 1996.

[43] A. Finkelstein. Requirements engineering: a review and research agenda. In *Proc 1st Asian & Pacific Software Engineering Conference*, pages 10–19. IEEE CS Press, 1994.

[44] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, Aug. 1994.

[45] R. F. Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32:221–233, June 1948.

[46] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication (RFC 2617). Technical report, The Internet Society, June 1999. (Available electronically at `http://www.ics.uci.edu/pub/ietf/http/rfc2617.txt`).

[47] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Strategies for incorporating formal specifications in software development. *Communications of the ACM*, 37(10):74–86, Oct. 1994.

[48] J. Goguen and C. Linde. Techniques for requirements elicitation. In *Proc. IEEE International Symposium on Requirements Engineering*, pages 152–164. IEEE CS Press, 1993.

[49] L. Goldin and D. M. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engineering*, 4(4):375–412, Oct. 1997.

[50] J. Grudin. *Groupware and Cooperative Work: problems and prospects.* Addison-Wesley, Reading, MA, 1990.

[51] J. Grudin and S. Poltrock. Computer supported cooperative work and groupware: A survey of systems and the behavioral and social issues that affect development and use. In *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, Tutorials, page 5, 1994.

[52] V. Hamilton. Experience of combining Yourdon and VDM. In *Proceedings of the Methods Integration Workshop*. Springer-Verlag, 1992.

[53] A. Hars. Advancing CASE productivity by using natural language processing and computerized ontologies: The ACAPULCO system. In *Proceedings of the Eleventh Automated Software Engineering Conference*, 1996.

[54] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 109–122, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[55] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadway. SCR*: A toolset for specifying and analyzing software requirements. In *Proceedings of the Tenth Annual Conference on Computer-Aided Verification (CAV'98)*, Vancouver, Canada, 1998.

[56] C. L. Heitmeyer, R. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[57] G. J. Holzmann. Proving properties of concurrent systems with SPIN. *Lecture Notes in Computer Science*, 962, 1995.

[58] W. S. Humphrey. *Managing the Software Process.* SEI Series in Software Engineering. Addison–Wesley, 1989.

[59] D. Jackson. Counterpoint: Requirements need form, maybe formality. *IEEE Software*, 13(2):21–22, Mar. 1996.

[60] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[61] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, Apr. 1996.

[62] P. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proc. of the 16th International Conference on Software Engineering*, pages 113–122, Los Alamitos, California, Mar. 1994. IEEE CS Press.

[63] W. L. Johnson, M. S. Feather, and D. R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, Oct. 1992.

[64] C. Jones. *A Short History of Function Points and Feature Points.* Software Productivity Research, Inc., Burlington, Mass., June 1986.

[65] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, New York, N.Y., 1991.

[66] M. Kao, N. Cercone, and W. S. Luk. Providing quality responses with natural language interfaces: The null value problem. *IEEE Transactions on Software Engineering*, 14(7):959–984, July 1988.

[67] M. Keil and E. Carmel. Customer-developer links in software development. *Communications of the ACM*, 38(5):33–44, May 1995.

[68] A. Kemmerer. Integrating formal methods into the development process. *IEEE Software*, 7(5):37–50, Sept. 1990.

[69] J. Krogstie, O. I. Lindland, and G. Sindre. Towards a deeper understanding of quality in requirements engineering. In *7th International CAiSE Conference*, volume 932 of *Lecture Notes in Computer Science*, pages 82–95, 1995.

[70] J. Krogstie, O. I. Lindland, and G. Sindre. Towards a deeper understanding of quality in requirements engineering. In *Proceedings of the Seventh International CAiSE Conference*, volume 932 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1995.

[71] P. G. Larsen, J. van Katwijk, N. Plat, K. Pronk, and H. Toetenel. SVDM: An integrated combination of SA and VDM. In *Methods Integration Conference*. Springer-Verlag, Sept. 1991.

[72] B. Lawrence. Point: Do you really need formal requirements? *IEEE Software*, 13(2):20, 22, Mar. 1996.

[73] D. B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11), Nov. 1995.

[74] D. B. Lenat, R. V. Guha, K. Pittman, D. Pratt, and M. Shepherd. Cyc: Toward programs with common sense. *Communications of the ACM*, 33(8), Aug. 1990.

[75] O. I. Lindland, G. Sindre, and A. Sølvberg. Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49, Mar. 1994.

[76] B. Macias and S. G. Pulman. *Natural Language Processing for Requirements Specification*, pages 57–89. Chapman and Hall, 1993.

[77] B. Macias and S. G. Pulman. A method for controlling the production of specifications in natural language. *The Computer Journal*, 38(4):310–318, 1995.

[78] N. A. M. Maiden. Reuse-oriented requirements engineering in NATURE. *ACM SIGSOFT Software Engineering Notes*, 20(3), July 1995.

[79] D. A. Marca and C. L. McGowan. *SADT : Structured Analysis and Design Techniques*. McGraw-Hill, New York, 1988.

[80] D. R. McCallum and J. L. Peterson. Computer-based readability indexes. In *Proceedings of the ACM'82 Conference*, pages 44–48, Oct. 1982.

[81] B. Meyer. Sur le formalisme dans le spécifications. *Globule, Newsletter of the AFCET WG on Software Engineering*, 1:81–122, 1979. ([82] is the later English version).

[82] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, Jan. 1985.

[83] S. Meyers and S. P. Reiss. A system for multiparadigm development of software systems. In J.-P. Finance, editor, *Proc. of the 6th International Workshop on Software Specification and Design*, pages 202–211. IEEE CS Press, Oct. 1991.

[84] G. P. Mullery. CORE : A method for controlled requirement expression. In *4th International Conference on Software Engineering*, pages 126–135, Long Beach, California, Sept. 1979. IEEE CS Press.

[85] G. P. Mullery. Acquisition-environment. In M. W. Alford, J. P. Ansort, G. Hommel, B. Lamport, B. Liskov, J. Mullery, and F. B. Schneider, editors, *Distributed Systems: Methods and Tools for Specification*, pages 45–130. Springer, Berlin, 1985.

[86] G. C. Murphy and D. Notkin. Lightweight source model extraction. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 116–127, Oct. 1995.

[87] NASA/Boeing. Software requirements specification for the NCS MDM CSCI. International Space Station Document S684-10174, Apr. 1997.

[88] P. Naur and B. Randell, editors. *Proc. of the NATO Working Conference on Software Engineering.* NATO Science Commitee, Oct. 1968.

[89] J. T. Nosek and R. B. Schwartz. User validation of information system requirements: Some empirical results. *IEEE Transactions on Software Engineering*, 14(9), Sept. 1988.

[90] B. Nuseibeh. Conflicting requirements: When the customer is not always right. *Requirements Engineering Journal*, 1(1), 1996.

[91] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, May 1997.

[92] B. Nuseibeh and A. Finkelstein. Viewpoints: A vehicle for method and tool integration. In *Proc. of the Int. Workshop on CASE (CASE'92)*, pages 50–60, Los Alamitos, California, July 1992. IEEE CS Press.

[93] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, Oct. 1994.

[94] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.

[95] K. Pohl. The three dimensions of requirements engineering: a framework and its applications. *Information Systems*, 19(3):243–258, 1994.

[96] K. Pohl and P. Haumer. HYDRA: A hypertext model for structuring informal requirements representations. In K. Pohl and P. Peters, editors, *Proc. of the* $2^{nd}$ *Int. Workshop on Requirement Engineering: Foundation of Software Quality (REFSQ 95)*, pages 118–134, Aachen, Germany, June 1995. Augustinus.

[97] H. B. Reubenstein and R. C. Waters. The requirements apprentice: An initial scenario. In S. Greenspan, editor, *Proceedings of the 5th International Workshop on Software Specification and Design*, pages 211–218, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[98] H. B. Reubenstein and R. C. Waters. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, Mar. 1991.

[99] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modelling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[100] A. Russo, B. Nuseibeh, and J. Kramer. Restructuring requirements specifications: a case study. In *Proceedings of the Second International Conference on Requirements Engineering*, Colorado Springs, USA, Apr. 1998.

[101] A. Russo, B. Nuseibeh, and J. Kramer. Restructuring requirements specifications. *IEE Proceedings: Software*, 1999.

[102] R. W. Schwanke and G. E. Kaiser. Living with inconsistency in large systems. In *Proc. of the International Workshop on Software Version and Configuration Control*, pages 98–118, Grassau, Germany, Jan. 1988.

[103] S. Shlaer and S. J. Mellor. *Object–Oriented Systems Analysis: Modeling the World in Data.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[104] G. Spanoudakis and A. Finkelstein. Overlaps among requirements specifications. In *Proc. of the ICSE Workshop on Living with Inconsistency*, Los Alamitos, California, 1997. IEEE CS Press.

[105] M. Spivey. *The Z Notation: A Reference Manual.* Oxford University Research Group, 1987.

[106] W. R. Swartout. GIST English generator. In D. Waltz, editor, *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982. AAAI Press.

[107] W. R. Swartout. The GIST behavior explainer. In M. A. Genesereth, editor, *Proceedings of the National Conference on Artificial Intelligence*, Washington, DC, 1983. AAAI Press.

[108] A. Taivalsaari. Multidimensional browsing. In *Proc. of the 8th Conference on Software Engineering Environments*, pages 11–22, Los Alamitos, California, Apr. 1997. IEEE CS Press.

[109] The Institute of Electrical and Electronic Engineers, Inc. Software engineering standards, 1997. (IEEE 610.12).

[110] P. Thomas. *CSCW Requirements and Evalution.* Springer Verlag, London, 1996.

[111] W. Tichy. RCS–a system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.

[112] D. Wood et al. A multimedia approach to requirements capture and modeling. In *Proc. of the* $16^{th}$ *International Conference on Software Engineering*, pages 53–56, Los Alamitos, California, Apr. 1994. IEEE CS Press.

[113] E. Yourdon. *Modern Structured Analysis.* Prentice–Hall, Englewood Cliffs, 1989.

[114] P. Zave. Call for papers and associated classification scheme. In *IEEE International Symposium on Requirements Engineering*, 1995.

[115] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodologies*, 2(4):379–411, Oct. 1993.