Università di Pisa
Dipartimento di Informatica

Technical Report: TR-03-05

The Circe approach to the systematic analysis of NL requirements

V. Ambriola V. Gervasi

March 14, 2003

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

The CIRCE approach to the systematic analysis of NL requirements

V. Ambriola V. Gervasi

March 14, 2003

Abstract

This paper presents CIRCE, an environment for the analysis of natural language requirements. CIRCE is first presented in terms of its architecture, based on a transformational paradigm. Details are given for the various transformation steps, including (i) a novel technique for parsing natural language requirements, and (ii) an expert system based on modular agents, embodying intensional knowledge about software systems in general.

Some of the features of the environment are shown by means of an example. Various stages of requirements analysis are covered, from initial sketches to pseudo-code and UML models.

Keywords: Natural language requirements, requirements analysis, software modeling, validation.

1 Introduction

The field of Requirements Engineering (RE) has witnessed remarkable progress in the last 25 years. Since it was first identified as an autonomous discipline, its role and importance in the software development process has never been confuted. From the first estimates [8] to recent studies [35], researchers agree almost unanimously that a carefully conducted requirements analysis is a key factor in the success of software development projects.

The increased maturity of the field, that is reflected in the evolution of the major RE symposia [16], has led to the structuring of the discipline into a number of well-defined sub-areas. [41] names *elicitation*, *modeling and analyzing*, *communication*, *agreement* and *evolution* of requirements as the "core" requirements activities. Other relevant activities are *domain analysis*, *specification* and *specification analysis* [51]. Many empirical and theoretical contributions have appeared in all those areas, but only in a few cases the research effort has been accompanied by a widespread adoption of *tools* and *environments* to support the various activities and processes. In particular, *requirements management tools* (e.g., Rational RequisitePro and Telelogic DOORS) and *specification analysis* *tools* (e.g., model checkers and SCR* [32]) are nowadays reasonably common among practitioners.

In this paper we present Circe, an environment supporting the second of the activities cited above, namely requirements modeling and analysis. Circe provides a simple yet powerful framework, and a working implementation of it, for analyzing requirements expressed in natural language (NL). Strong emphasis is placed on the understandability of the requirements, in order to facilitate discussion with the customer. To this end, a non-standard approach to natural language parsing is adopted, capitalizing on the peculiarities of software requirements. The modeling and analysis activity is performed by a modular expert system, guaranteeing easy extendibility and customizability of the environment.

Under many respects, our proposal brings the analysis of natural language requirements into the time-honored tradition of transformational approaches, that has been proved so effective in the field of compiler construction. At the same time, we emphasize continuous and immediate interaction between the tool and the requirements engineer (involving, possibly, also the customer), in recognition of the fundamental role that human professionals play in this activity.

This paper focuses on the Circe environment from an architectural point of view, and intentionally does not discuss process-related issues. It is our belief that tools should be general and adaptable enough to be used in different type of processes, rather than the opposite. This leaves requirements engineers free to choose the process that is most suited to a particular task, and still be able to use their toolbox, rather than forcing the same tool-defined process on all problems. However, we provide motivations and an example case showing how Circe can be applied in a variety of different situations.

The paper is structured as follows. After introducing some general motivation in Section 2, a general overview of Circe's architecture is provided, identifying the main transformation steps involved in the synthesis and analysis of models from the text of the requirements. The various steps are then described in a certain detail in Sections 4 (concerning the parsing of NL requirements) and 5 (concerning model building and visualization). In Circe, requirements are interpreted according to a basic model of how software is built and works; we call this model the Circe Native Meta-model, or CNM for short. The CNM is briefly discussed in Section 6, which provides a high-level description of the main concepts involved. Finally, a hands-on example of how Circe can be used in a real case is presented in Section 7, that shows (for a fragment of a simple embedded control software) how analysis and model synthesis is conducted with Circe, from NL requirements all the way up to UML models and pseudo-code. The example also shows some of the validation features of the environment. A short survey of related work and some conclusions complete the paper.

2 A motivating scenario

Let us suppose we are asked to collect and analyze the requirements for an in-flight missile control system. We could start by writing down a few simple statements describing the software, the part of the world with which it will interact, and its expected operations. This first description could include a statement like

> After take off, the FMCS reads the height from the altimeter and the heading from the gyroscope every 20 milliseconds. It computes appropriate navigation commands, based on the current target position, and sends these commands to the engine controller.

Since we are working systematically, we also write down a list of the terms and abbreviations that we are using, e.g.

> **FMCS:** the Fictitious Missile Control System.
> **altimeter:** a sensor measuring height from ground.
> **gyroscope:** a sensor measuring heading and inclination relative to an arbitrary initial setting.

Given this information, an intelligent requirements analysis tool should already provide the user with some helpful hint. For example: who defines the "current target position"? Or, what navigation commands are "appropriate"? Providing alternate representations of the requirements is also helpful, and can expose other problems. A good tool could extract and present a data flow model from our requirements, like the one shown in Figure 1. Looking at that graphical representation, the customer or the requirements engineer could realize that navigation commands should also take height and heading into consideration, and possibly other information as well. Requirements should be added to correct these and other problems, and more requirements to describe other functions of our FMCS could be written.

As our requirements base grows, more and more information becomes available. An intelligent tool should be able to extract the information from the requirements and reorganize it into other forms, more suitable for specific tasks. We could also ask for UML models that are automatically synthesized from the requirements (like, for example, those in Figure 2).

Moreover, once we have entrusted our requirements to a tool, we could also ask for process control embedded in the tool. We could then obtain answers to questions like: Have our requirements been changing a lot lately? What new functions are stable enough to warrant inclusion in the next stable release? Are errors being kept under control? What is our current function points estimate?

All these questions could be answered by tracing the evolution of the requirements over time. As an example, consider the diagram in Figure 3: the spikes in the functional and behavioral complexity scores warn us that the requirements are still *substantially* in a state of flux — i.e., editing on the requirements is not cosmetic —, and the relatively high number of unresolved errors and warnings constitutes a serious indication that the process may not be managed well.

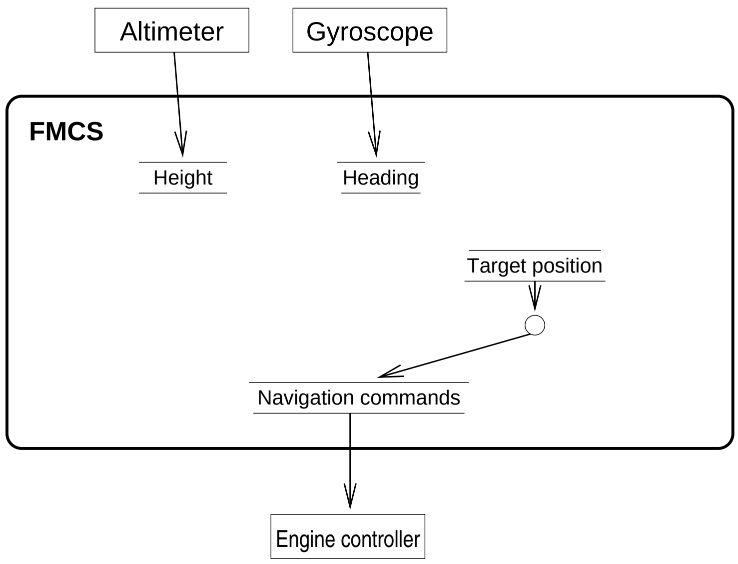


Figure 1: A data flow model extracted from the sample requirements for the FMCS.

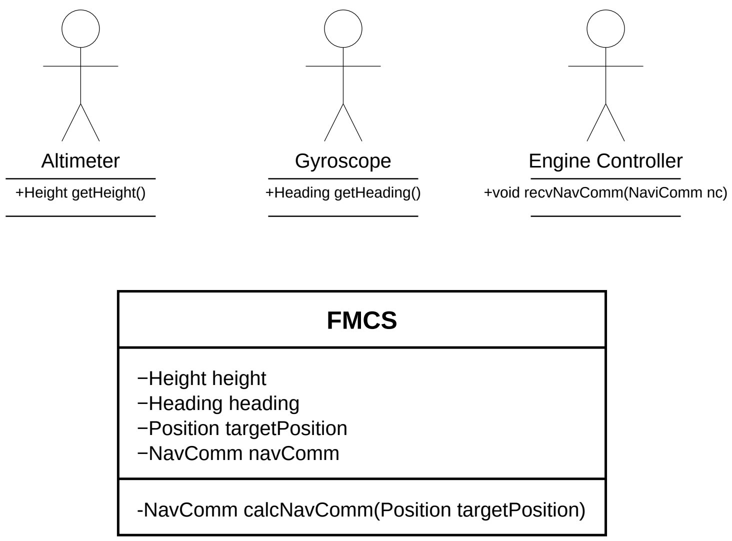


Figure 2: UML object diagram extracted from the sample requirements for the FMCS.

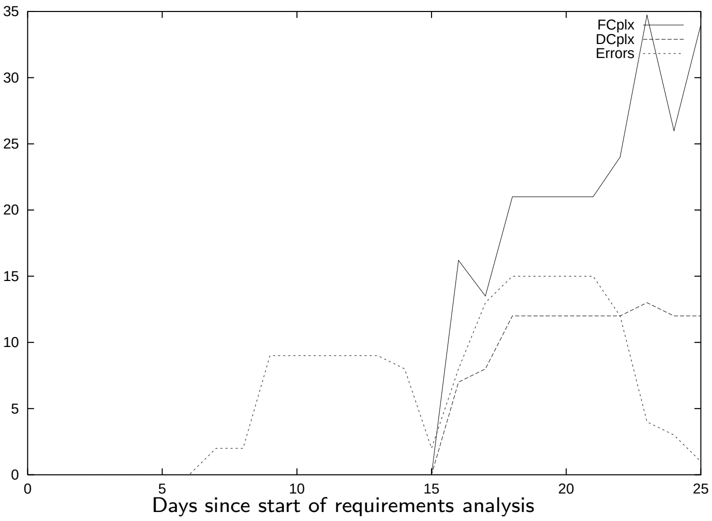


Figure 3: Process diagrams for a requirements development process. Plotted in the diagram are measures of functional complexity (FCplx), data complexity (DCplx) and the number of errors found in the requirements.

In this scenario, a large number of different technologies are needed to work out the "magic": analysis of natural language text, synthesis of software engineering models, analysis of those models, expert systems to express judgments over the results of the analysis, quality models and metrics, and so on. In the following, we will present the general framework and the specific technologies that our tool CIRCE employes to provide to its users the very same services that we found desirable in our discussion above.

3 General overview

CIRCE is an environment for the analysis of natural language requirements. It is based on the concept of successive *transformations* that are applied to the requirements, in order to obtain concrete (i.e., rendered) *views* of *models* extracted from the requirements. These include:

- models of the *requirements document* itself, considered as a textual artifact,
- models of the *system described by the requirements*, as in Figures 1 and 2 from our previous example, and
- models of the *requirements development process*, through the traces of relevant artifacts, as in Figure 3 from our example.

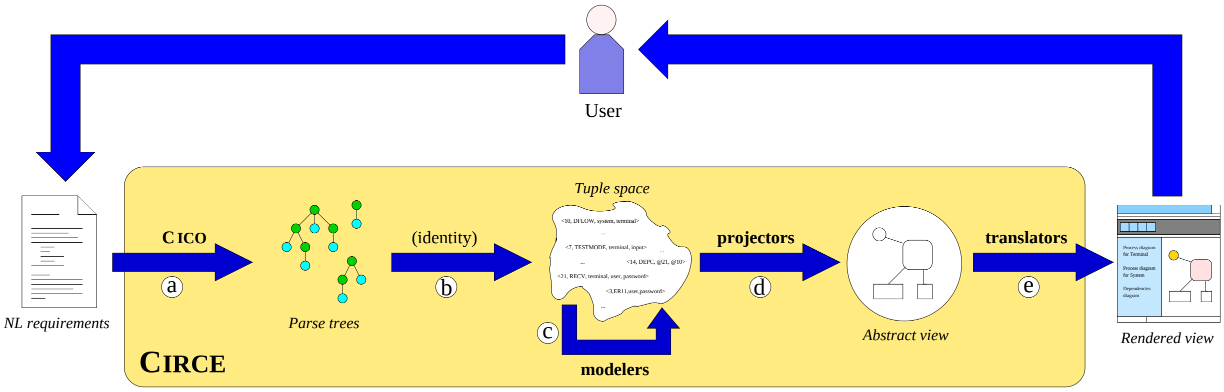


Figure 4: A general overview of the transformations on NL requirements operated by CIRCE.

The general architecture of CIRCE is shown in Figure 4. The whole transformation process (from NL requirements to concrete views of models) is divided into five steps, labeled ⓐ-ⓔ in the figure. Each of the steps can be considered as a function with the domain and co-domain depicted in the figure, and in the subsequent sections they are formally specified as such (although some implementation notes are also given). The steps are briefly described in the following:

ⓐ First, the natural language text of the requirements is *parsed* and transformed into a forest of parse trees. These parse trees closely correspond to the original requirements, but abstract away many surface features, thus facilitating subsequent analysis. For this transformation, CIRCE uses a domain-based parser called CICO. The algorithms and techniques employed for this step are fully described in Section 4.

ⓑ The parse trees obtained from the previous transformation are then encoded as tuples and stored in a shared tuple space. In CIRCE, this encoding is performed by CICO alongside with the parsing process: as a consequence, this step can be considered formally to consist in an identity transformation. These tuple-encoded parse trees provide the extensional knowledge about the requirements, and serve as basis for the next step.

ⓒ An embedded expert system is then called upon to refine the extensional knowledge in the tuple space, enriching it with more tuples. These are derived from those produced in ⓑ and from the intensional knowledge about the basic structure and behavior of software systems that is provided by modular components ("experts") called *modelers*. CIRCE includes a library of over one hundred modelers, covering such diverse aspects as static, functional and behavioral modeling, validation and metrication of such models, document structure analysis, synthesis of user interface components, etc. The details of this transformation are described in Section 5.1.

ⓓ When a specific *view* on the requirements is desired, the needed information is extracted from the shared tuple space by a second class of components called *projectors*. This transformation produces an abstract (e.g., graph-theoretic) description of the desired view, starting from the extensional and intensional knowledge collected from the previous steps. This abstract view still exists only as an internal representation in CIRCE, and needs to be made concrete and offered to the user.

ⓔ Making the abstract views concrete is the purpose of the last transformation. The abstract view is taken as input by other components called *translators*, and actual rendering is performed, thus producing a concrete view. Several different rendering modes may be offered for the same abstract view, to accommodate for different usage contexts. For example, a graph can be rendered either with interactive placement of nodes and zooming, to facilitate browsing, or as a static picture, for inclusion in paper documents. This last transformation, together with the previous one, is described in Section 5.2.

From a user perspective, CIRCE is a work companion that reads natural language requirements as input, and — upon user's request — produces a vast number of views on different aspects of the requirements. The user can leverage the insights gained from the various views to correct, complete or perfect the original NL requirements. This induces a loop that includes the human as a driving force for the requirements development process, while offering him powerful tools to analyze the requirements in great detail.

CIRCE has been implemented as a web-based system. A central server acts as a repository for requirements documents. By using a standard web browser, users can edit their requirements, and ask for specific views. For each request, CIRCE performs the transformations[1] ⓐ-ⓔ needed to generate the requested view, and sends back the result as a web page (possibly enriched with active content: for example, by using Java applets for interactive graph browsing). Users need no special software installed.

A sample session with CIRCE is shown in Figure 5. Section 7 includes other screenshots of CIRCE's user interface.

4 Linguistic analysis

In order to precisely define the first of the transformation steps outlined in the previous section, we must first provide a rigorous definition of our input artifact (a requirements document) and output artifact (a forest of parse trees). We then present in detail the specific parsing algorithms that are used in CIRCE.

[1]Pervasive caching and differential parsing are used to improve performance.

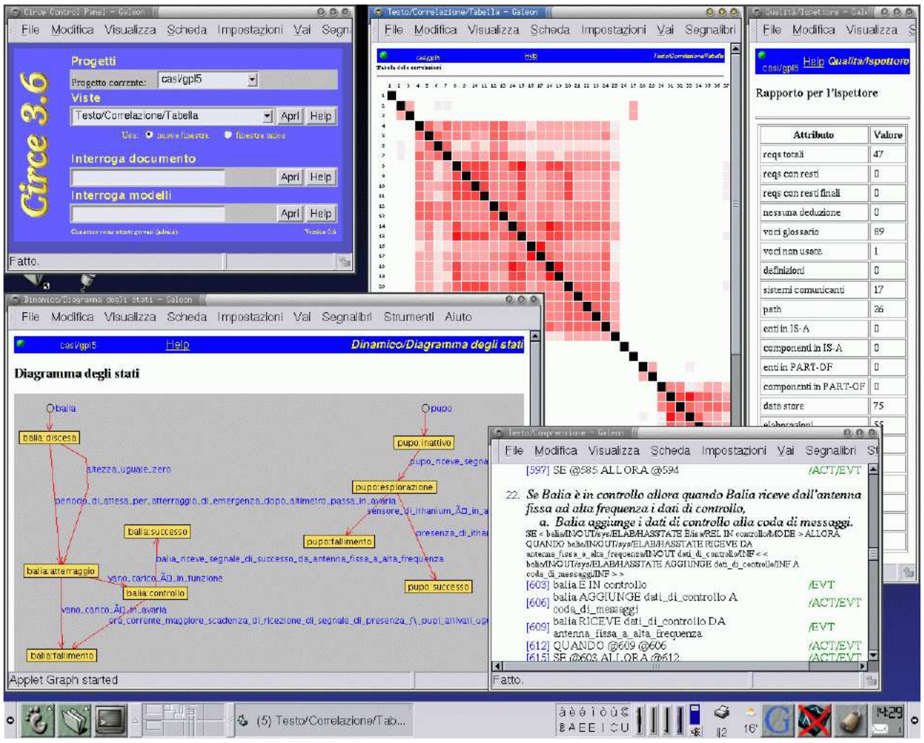


Figure 5: A sample session with Circe. The figure also shows the multilingual support in Circe: Italian is used in this case. See Section 7 for more screenshots in English.

4.1 Requirements document model

In CIRCE, a requirements document $\mathcal{D}$ is a triple $\langle \mathcal{G}, \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{G}$ is a set of *designations* (also called a *glossary*), $\mathcal{F}$ is a set of *definitions*, and $\mathcal{R}$ is a set of *requirements*.

In keeping with the terminology by Jackson [34], designations in $\mathcal{G}$ establish specific *names* for entities or phenomena that are relevant in the domain being considered. Terms in $\mathcal{G}$ are *tagged* with labels declaring their properties. In addition, designations explicitly establish equivalence between different terms (e.g., synonyms or metonyms) that refer to the same entity.

Definitions, on the other hand, establish notational shorthands for expressing requirements in a succinct and practical way. Definitions in $\mathcal{F}$ allow the requirements writer to use the language that is most appropriate to the domain, while at the same time declaring in precise terms how domain idioms should be interpreted.

Finally, requirements in $\mathcal{R}$ describe the expected behavior of the system. This description may include details on the environment (to make it clear how the system affects its environment and how is affected by it) and on the internal structure of the system itself (to accommodate for architectural constraints like scalability, performance, compatibility, etc.).

As an example, consider the following requirement for our fictitious missile in-flight control system:

Requirements

```
When the system receives a launch confirmation command from
Flight Control, it shall turn the main engine on within 0.8
seconds.
```

Terms like "system", "launch confirmation command", "Flight Control", and "main engine" are domain-specific, and designate certain physical or logical entities that are relevant in a particular domain. Thus, these terms should be explicitly declared in $\mathcal{G}$. On the other hand, common words such as "When", "receive", "from", "within", and "seconds" can be considered well-known, and in fact their linguistic role is known to the parsing system (by virtue of being used in the parsing rules, as we will see shortly).

For example, in CIRCE communication of data and commands among active entities is a primitive concept, thus the statement "the system receives a launch confirmation command from Flight Control" needs no further explanation. However, since the concept of turning things on or off is not primitive in CIRCE, we have to provide an explicit definition for it. In our case, the definition could be simply

---

Definitions

```
TURN x ON  ↦  SEND AN ACTIVATION COMMAND TO x
```

---

where the $\mapsto$ (read: "is defined as") symbol indicates that any occurrence of the left part in the requirements may be replaced by the right part. In this notation, parts in upper case denote constants that must match literally, while words in lower case denote free variables that can match intervening text (and bind to the matched text in the replacement). Details about what constitutes a matching are presented in Section 4.3.2.

Designations and definitions can be applied to requirements in order to obtain a *canonical form* for the requirements. In our previous example, assuming that "LCC" and "base" have been declared as equivalent (and preferred) designation for "launch confirmation command" and "Flight Control", respectively, and that "activation command" has been declared in $\mathcal{G}$, we obtain the canonical requirement[2]

Requirements

```
When the system receives a LCC from base, it shall send an
activation command to the main engine within 0.8 seconds.
```

To summarize, in $\mathcal{D}$ the application of $\mathcal{G}$ and $\mathcal{F}$ to $\mathcal{R}$ has the effect of transforming a requirement stated in the customer's language into a canonic form that only refers to primitive terms and concepts. These canonic requirements can be analyzed more easily in a systematic way, and any assumption made in the canonization process is cleanly recorded in the $\mathcal{G}$ and $\mathcal{F}$ artifacts.

The structuring of requirements documents into $\mathcal{G}$, $\mathcal{F}$, and $\mathcal{R}$ has the added benefit that $\mathcal{F}$ records explicitly those parts of the domain knowledge needed during the transformation of the user requirements into a specification for the software, as well as any design decision made in the process.

In our previous example, the definition

Definitions

```
TURN x ON  ↦  SEND AN ACTIVATION COMMAND TO x
```

that we presented as a convenient way of introducing a shorthand for "turning something on", can also be considered as (i) an explanation of what the user meant for "turning something on", if we assume that the on/off mechanism is already implemented, and thus is part of the domain, or (ii) as a design decision we have made, if we assume that we are in charge of implementing the on/off mechanism for the engine ourselves. We can discriminate between these two uses by looking at whether the subsystem x that we are considering is part of the system we are going to design or not.

4.2 Tokenization and morphosyntactic analysis

While designations and definitions are structured documents, which must conform to a formal — albeit simple — syntax, requirements and domain description statements are expected to be essentially free-form text. The first step in our linguistic analysis consists of handling typographical and formatting details

[2]For clarity, we have omitted in this example the tags that would be appended to the various designation.

of the text, in tokenizing it by identifying its basic components (words, morphemes, punctuation characters, etc.), and in assigning part-of-speech and other appropriate labels, or *tags*, to each token.

Many of these operations are relatively trivial, but nevertheless essential to make the system usable even to untrained users, who can adopt a variety of styles and conventions in preparing their input documents. This *canonization* process, that is preliminary to parsing proper, is described in the next sections.

4.2.1 Typographical adjustments

Many human languages tend to have alternative 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 as `ss`; accented vowels in Italian are sometimes written as a simple vowel 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, a first transformation step performs typographical canonization 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.

The English language needs a comparatively simpler treatment, including

- splitting genitives from the noun they are attached to, to simplify subsequent analysis (e.g., `message's` → `message 's`);
- splitting and expanding contracted forms of certain verbs and negation (e.g., `can't` → `can not`);
- substituting various quotation styles with the uniform use of double quotes.

We are currently in the process of extending the system to also accept Japanese text, that will introduce the need for further processing (e.g., segmentation).

4.2.2 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 white space characters to a single space;
- discarding comment lines;[3]
- 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 4.3.8);
- transforming quoted excerpts into a single term;
- discarding remaining unneeded punctuation characters.

At the end of this stage, the input stream consists of a sequence of statements, one per line; each statement consists of a sequence of words and special markers, separated by spaces. Formally, a requirement r at this stage is defined as a sequence of words ω_i:

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

4.2.3 Designation handling

As we noted above, designations serve two purposes: they declare properties of the term being designated, by annotating each term with a set of semantic *tags*, and establish a map from any synonym to the canonical form of a term. Formally, a set of designations $\mathcal{G}$ is defined as a set of term-synonyms pairs:

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

where τ is the base term (i.e., the canonical form for the term) and τ_i are synonyms for τ. In the case of our example, the set of designations could be

$$\begin{aligned} \mathcal{G} \quad = \{ \quad & (\text{LCC/INF}, \{\text{Launch Confirmation Command}\}), \\ & (\text{base/IN/OUT}, \{\text{Launch Control Center}, \text{Launch Control}\}), \\ & (\text{main engine/IN}, \{\text{main propeller}, \text{propeller}\}), \\ & (\text{system/IN/OUT/ELAB/SYS/HASSTATE}, \{\}) \\ & \} \end{aligned}$$

[3]Information linking comments to the place in which they appeared is saved, making it possible to retrieve comments pertaining to a certain requirements in later processing for visualization purposes.

Given a term τ, we define the two functions $\mathsf{term}(\tau)$ and $\mathsf{tags}(\tau)$ to be, respectively, the base form of the term and the set of its tags. For example, from the designations above we have

$$\begin{aligned}\mathsf{term}(\text{base/IN/OUT}) &= \text{base}\\ \mathsf{tags}(\text{base/IN/OUT}) &= \{/\text{IN},\ /\text{OUT}\}\end{aligned}$$

Glossary substitution is obtained by applying the $\mathsf{map}_{\mathcal{G}}()$ function defined below to each ω_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:

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

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

In informal terms, the first condition states that a term must not appear more than once in a glossary, while the second ones states that the same word cannot be declared as synonym to two different terms.

Notice that these uniqueness conditions are only requested for terms explicitly introduced as designations, i.e., it is required that the name assigned to each distinct entity (or class of entities) in the domain is unique.

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

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

4.2.4 Morphosyntactic analysis

Morphosyntactic analysis on the text is performed with the help of Helmut Schmid's TreeTagger [50], a part-of-speech tagger based on probabilistic decision trees obtained from annotated corpora. Given a sequence of tokens representing a requirements, this step annotates each word with a set of tags describing its part-of-speech role, and where applicable substitutes the token with the base form (lemma) of the word. For English, we use a variant of the standard Penn Treebank tagset [38], in which complex tags (e.g., `/NNP` standing for proper noun) have been split according to their components (e.g., `/NN/P` standing for "a noun, a proper one") in order to permit matching on whole classes (e.g., a `/NN` tag would match a name, of any kind).

For example, our sample requirement from page 10, after all the steps up to morphosyntactic analysis, would appear as follows:

```
when/WRB the/DT system/NN/IN/OUT/ELAB/SYS receive/VB/Z
a/DT LCC/NN/INF from/IN base/NN/IN/OUT ,/, it/PP
shall/MD send/VB an/DT activation_command/NN/INF to/TO
the/DT main_engine/NN/IN within/IN 0.8/CD second/NN
```

(for clarity, we have underlined the tags explicitly introduced in designations; all other tags are produced by TreeTagger).

4.3 Parsing

Given a text tagged and canonized as shown above, CIRCE parses the text by using the CICO domain based parser [1, 27, 28].

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

4.3.1 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 ρ is defined as

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

where

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

In the following we will refer to the model part of a MAS rule as the *template*, to avoid ambiguity with other uses of the term "model". The syntactic denotation for a MAS rule is

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

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 ρ applied to a statement r is "if the template μ of ρ matches a fragment of r, execute the action α and substitute the matching fragment with σ".

Terms in a rule are either constants or variables; the function $\mathsf{isvar}(\tau)$ is true if τ is a variable, and $\mathsf{isconst}(\tau)$ is true if τ 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 μ. This binding is then used when the rule is fired to assign values to the variables in α and σ.

4.3.2 Matchings

Given a term τ and a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$, we define a set of possible *targets* for τ 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 template of ρ, 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 template 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, the first element of which is described below.

4.3.3 Match scoring

Since templates express "natural" schemas for sentence fragments, the ideal match for a template would be a one-to-one, ordered correspondence between elements of the template and terms in a statement. However, natural languages usually permit a certain degree of freedom in the placement of words. This is particularly true of those languages — like Italian or French — that permit 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 be discarded when analyzing a requirement 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 template 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.

Tuning of these parameters depends on the desired strictness for the parsing process, or, in other terms, on the level of formality that is desired in the requirements text. High values of $K_{DROPPED}$, K_{DIST}, and $K_{INVERSION}$ make the parser very strict on how the rules can be applied. Thus, if an organization has a policy calling for a well defined and precise language to be used in requirements, CICO can be used as a syntax checker for a formal language by choosing appropriate values for the parameters.

On the other hand, if ease of change, rapid development, and freedom of experimentation are the main goals, low values of these parameters make CICO behave like an information extraction tool. In this case, the parser takes an optimistic view on the syntactic correctness of the requirements. Rather than checking if the language used in the requirements strictly follows the syntax expressed by the MAS rules in use, the parser tries to extract as much significant information as possible from the requirements, possibly skipping over unrecognized passages and ignoring textual inversion and other similar phenomena. CICO's flexibility makes CIRCE adaptable to different development practices — a characteristic that we value as an important positive feature of the environment.

4.3.4 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 sequence of terms r, a template $\mu = \langle \tau_1, \dots, \tau_n \rangle$, a statement $s = \langle \omega_1, \dots, \omega_m \rangle$ and a valid match $m = \langle t_1, \dots, t_n \rangle$ between μ and s, we define[4]

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

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

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

where

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

To characterize the application and firing of a rule $\rho = (\mu, \alpha, \sigma)$ to a statement s, we first define a statement-action structure (briefly, a *sa-structure*) as a pair (s, A) where s is a statement and A is a sequence of actions. The function[5]

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

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

4.3.5 Rule scoring

Beside the score due to the ordering of matches, a second score is computed depending on the *specificity* of a rule. We define specificity of a rule $\rho = (\mu, \alpha, \sigma)$, where $\mu = \langle \tau_1, \dots, \tau_n \rangle$, as the sum of three factors:

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

Moreover, we assign a fixed score to each rule to favor longer chains of rule applications (see Section 4.3.7). We define then

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

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

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

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

4.3.6 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 next section) 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 negative score is given to each rest, computed as

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

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

4.3.7 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 guarantees that a unique interpretation is returned by the parsing process. For validation purposes, in CIRCE the parsing sequence is also presented back to the user for confirmation. Nevertheless, it would be possible and sensible in different contexts to output all the possible parsing sequences, or only the best-scoring ones, and leave to further processing the task of selecting the "right" parsing sequence for a given context.

4.3.8 Parsing contexts

As introduced in Section 4.2.2, 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**
  $B_k := \textsc{VeryBadScore}$
  BasicParse($r_{|\mathcal{G}}, R, \langle\rangle, 0$)
  if $B_k \geq \textsc{MinimumAcceptableScore}$ **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

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

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 indices 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.

Algorithm 3 ContextParse(s, R).

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

4.3.9 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.

It is important 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, 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 α_2 and α_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 ρ_1, leading to a rest $\langle S, C\rangle$, or in the application of ρ_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 ρ_3, thus causing the execution of α_2 and α_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).

4.4 Definition handling

We can now revisit the concept of *definition* that we introduced in Section 4.1. We recall that the purpose of a definition is to provide an interpretation rule for a new linguistic form, or to define a shorter form for a complex expression.

Both functions can be well served by transforming definitions into a simplified version of MAS rules. Intuitively, a definition can be seen as a MAS rule with no action part, where the template and substitution parts are obtained from the corresponding parts of the definition by applying the canonization steps described in Section 4.2 (including designations handling).

In formal terms, a definition δ

$$\mu \quad \mapsto \quad \sigma$$

is transformed into a MAS rule ρ given by

$$\rho = \langle \bar{\mu}, \langle\rangle, \bar{\sigma} \rangle$$

where the $\bar{\mu}$ and $\bar{\sigma}$ sequences are obtained from the corresponding μ and σ sequences of the definition (after undergoing typographical adjustment and format handling) by applying the *partial map* function $\overline{\mathsf{map}}_{\mathcal{G}}()$ defined below to each word ω in μ and σ:

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

Given a sequence $s = \langle \omega_1, \ldots, \omega_n \rangle$ we denote its partially-substituted version with $\bar{s}_{|\mathcal{G}}$:

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

Hence, given a glossary $\mathcal{G}$ and a set of definitions $\mathcal{F}$, we transform the definitions into a set of MAS rules $\bar{R}$ given by:

$$\bar{R}(\mathcal{G}, \mathcal{F}) = \{ \langle \bar{\mu}_{|\mathcal{G}}, \langle\rangle, \bar{\sigma}_{|\mathcal{G}} \rangle \mid \quad \mu \mapsto \sigma \in \mathcal{F} \}$$

In Circe, the parsing is performed by using a set of MAS rules obtained by union of rules from a standard library (see Section 6) and of the rules translated from the definitions (see Algorithm 5). This strategy allows the free interplay of parsing rules and definitions in the course of the parsing. Not only a definition can be applied on a text that is the result of some other definition or of other basic parsing rules, but application of the definition is itself *opportunistic.* Cico will apply the definition only where it makes sense (in terms of improved parsing score). Moreover, the same specificity-based strategy that is used for the basic rules, is applied also to definitions. This allows the user to provide multiple, overriding definitions. As an example, a definition could state that "notifying a human" means sending an e-mail to his or her account:

Definitions

```
NOTIFY x/HUMAN  ↦  SEND E-MAIL TO x'S ACCOUNT
```

But a second, more specific definition could instead state that "notifying the operation manager" (where the operation manager is a human) means sending a message to his or her pager:

Definitions

```
NOTIFY OP-MANAGER  ↦  SEND A MESSAGE TO THE OP-MANAGER'S
PAGER
```

In this case, the second definition would be transformed into a MAS rule that will be more specific than the one produced from the first definition, thus favoring the application of the second definition where the operation manager is to be notified.

Algorithm 5 FullParse($\mathcal{R}, \mathcal{G}, \mathcal{F}, R$).

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

4.5 Special forms of tagging

Five special forms of tagging exists, to support some advanced features of CIRCE. *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. *Wildcard tagging* supports template terms that match any tagged term in the statement — regardless of the specific tags it has. A special *optionality* tag indicates that if no matching is found for a certain term, the rule is still applicable, although a small penalty is imposed for missing terms. Finally, normal tags can be expressed in *negative* form, to indicate that certain tags should be removed from a term upon substitution.

All these forms of tagging are described individually in the following sections.

4.5.1 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 Section 4.3.2), if $\texttt{/?} \in \mathsf{tags}(\omega)$ 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

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

Next, we introduce the special action $\alpha^?(\omega, T)$ whose purpose is to record that term ω is being speculatively assigned all the tags in the set T. Given a template $\mu = \langle \mu_1, \ldots, \mu_n \rangle$, 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

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

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

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

where

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

As an example of application of speculative tagging, consider the rule $\rho = (\langle x, y/\mathrm{TAG}_1, C\rangle, \langle\alpha_0\rangle, \langle S\rangle)$[6] 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,

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

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 intended with respect to the scoring system and to the rule set in use, for the purpose of maximizing parsing score.

4.5.2 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,

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

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 template $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, a statement $s = \langle \omega_1, \ldots, \omega_m \rangle$, and a valid match $m = \langle t_1, \ldots, t_n \rangle$ between μ and s, we define

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

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 ρ 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 $\texttt{A/T}_1\texttt{/T}_2$ 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:

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

Similarly, the computation of the substitution yields

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

4.5.3 Wildcard tagging

Wildcard tagging, denoted by a `/*` tag, is a way to specify that a term in the template 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 template 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 $\mathsf{rscore}()$:

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

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).

4.5.4 Optional terms

Optionality of a term in the template part of a rule is indicated by a `/O` tag. When a term in a template has this tag, the rule can be fired even if no match for the term can be found. To formally define this mechanism, we introduce a special term representing "no match", that we denote with $\perp$. The target for an optional term can thus be any standard term, or $\perp$:

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

where Opt is defined as follows:

$$\mathit{Opt} = \begin{cases} \{\perp\} & \text{if } \texttt{/O} \in \mathsf{tags}(\tau) \\ \{\} & \text{otherwise} \end{cases}$$

Moreover, we must refine *Std*, *BRef*, and *FRef*, in order to ignore the `/O` tag in the template term when checking for tag inclusion:

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

where the new operator $\sqsubseteq$ on tag sets is defined as

$$t_1 \sqsubseteq t_2 \quad \stackrel{def}{\Longleftrightarrow} \quad (t_1 \setminus \{\texttt{/O}\}) \subseteq t_2$$

Finally, we must specify how $\perp$'s should be handled in substitutions. To this end, it is sufficient to refine the $\mathsf{ref}()$ function that we introduced in Section 4.5.2 so that the empty string[7] ϵ is substituted when an optional term has no match:

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

Valid matches (from Section 4.3.2) are also naturally extended to accommodate multiple $\perp$'s. Given a complete matching map $M = \langle T_1, \ldots, T_n \rangle$, a *valid match* m from M is defined as follows:

$$\begin{array}{lll} m = \langle t_1, \ldots, t_n \rangle \text{ s.t.} & \forall i = 1..n, & (t_i \in T_i) \quad \wedge \\ & & (t_i = \perp \quad \vee \quad \forall j = 1..n, t_i = t_j \Rightarrow i = j) \end{array}$$

[7]In the implementation, the empty string is represented by an underscore character in action parts. An explicit representation is needed to ensure that the arguments' positions in the output do not vary depending on whether an optional term was matched or not. Notice however that in substitution parts, empty strings are used for ϵ instead.

To reflect the intuition that having a match is generally preferable than having no match at all, even for optional terms, we also update the definition of $\mathsf{mscore}()$:

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

where O_m is the number of $\perp$'s in m, and K_{OPT} is a parameter of the cost model.

4.5.5 Negative tags

When performing a substitution, the $r\left[{}^{a}\!/\!_{b}\right]$ operator substitutes every instance of a in r with b, where a and b are terms, and r is a sequence of terms. To be more precise, a is replaced by the base term of b, with tags obtained by merging the tags of a and b. Formally,

$$\begin{aligned}\langle\rangle\left[{}^{a}\!/\!_{b}\right] &= \langle\rangle \\ (\langle x\rangle.r)\left[{}^{a}\!/\!_{b}\right] &= \langle x\rangle.(r\left[{}^{a}\!/\!_{b}\right]) \text{ if } x \neq a \\ (\langle a\rangle.r)\left[{}^{a}\!/\!_{b}\right] &= \langle c\rangle.(r\left[{}^{a}\!/\!_{b}\right])\end{aligned}$$

where c is defined by $\mathsf{term}(c) = \mathsf{term}(b)$ and $\mathsf{tags}(c) = \mathsf{tags}(a) \cup \mathsf{tags}(b)$. While this is generally the desired behavior, in some case (especially when simply rewriting fragments of text), it may be preferable to alter the set of tags of a substitution in a more flexible way. To this end, we define *negative tags*, as a way to remove specific tags from the resulting substitution.

Negative tags are denoted by inserting a minus sign between the slash and the name of the tag. Thus, the negative form of a tag `/TAG` is `/-TAG`. Negative tags are allowed only in the action and substitution part of a MAS rule, thus matching and scoring do not need any modification to support this kind of tags. Actually, only the basic replacement operator defined above needs to be refined. In particular, the replaced instance c is now defined by

$$\begin{aligned}\mathsf{term}(c) &= \mathsf{term}(b) \\ \mathsf{tags}(c) &= (\mathsf{tags}(a) \setminus \mathsf{tags}^-(b)) \cup \mathsf{tags}^+(b)\end{aligned}$$

where $\mathsf{tags}^-(b)$ is the set of the negative tags among the tags of b, and $\mathsf{tags}^+(b)$ is the set of the positive (normal) tags among the tags of b. Obviously, $\mathsf{tags}(b) = \mathsf{tags}^+(b) \cup \mathsf{tags}^-(b)$.

As negative tags can only be used in actions and substitutions, they can never occur in the source text (i.e., in the glossary), nor in the template parts of MAS rules.

As an example of application of this kind of tagging, consider the statement s and the rule ρ defined by

$$\begin{array}{rcl} s & = & \langle \mathtt{A/T_1/T_3, B, C/T_2} \rangle \\ \rho & = & (\langle x/T_1, B, y\rangle, \langle S, x/T_4/-T_3, y-T_4\rangle, \langle U, y/-T_2\rangle) \end{array}$$

The template part of the rule will match the text s, and assuming that no other rule in the current rule set can be applied, ρ is fired. In particular, x is bound to $A/T_1/T_3$, and y to C/T_2. In the action part, $/T_4$ is added to x, and T_3 is removed, thus obtaining $A/T_1/T_4$. Analogously, $/T_4$ is removed from y, but since the term bound to y did not have $/T_4$ anyway, the result is simply C/T_2. The final action is thus $\langle S, A/T_1/T_4, C/T_2\rangle$. The same process is applied to the substitution part, producing $\langle U, C\rangle$.

4.6 A parsing example

To see how the parsing algorithms described above works in practice, let us consider the following statement, again from the requirements for our FMCS:

```
The system shall send a heartbeat to the
base every 10 seconds.
```

Typographical adjustment and format handling modify the statement only in a slight way (e.g., to put it all in one line). Designation handling substitutes canonic forms of the designations for their various forms, and finally morphosyntactic analysis annotates the text with the relevant part-of-speech tags, thus giving[8]

$s=$ `the/DT system/NN/SYS/IN/OUT/ELAB shall/MD send/VB`
`an/DT heartbeat/NN/INF to/TO the/DT base/NN/IN/OUT`
`every/DT 10/JJ/CD second/NN/S`

The parsing step requires a library of MAS rules. For simplicity, let us assume that only the following rules are defined:

	Model	**Action**	**Substitution**
ρ_1	snd/OUT SEND data/INF TO recv/IN	SEND $snd $recv $data	$ID/ACT/EVT
ρ_2	EVERY period/TSPAN action/ACT	RPTACT $period $action	$ID/ACT
ρ_3	number/CD SECOND	TSPAN $number SECS	$ID/TSPAN

The matching maps for the various rules are as follows:

$$\begin{array}{rcl} \mathsf{match}(\rho_1, s) & = & \langle \underbrace{\{system, base\}}_{snd}, \underbrace{\{send\}}_{SEND}, \underbrace{\{heartbeat\}}_{data}, \underbrace{\{to\}}_{TO}, \underbrace{\{system, base\}}_{rcv} \rangle \\ \mathsf{match}(\rho_2, s) & = & \langle \underbrace{\{every\}}_{EVERY}, \underbrace{\emptyset}_{period}, \underbrace{\emptyset}_{action} \rangle \\ \mathsf{match}(\rho_3, s) & = & \langle \underbrace{\{10\}}_{number}, \underbrace{\{second\}}_{SECOND} \rangle \end{array}$$

[8]Actually, in the sake of efficiency the text is also tokenized, so that in subsequent phases word and tags matchings can be performed by simple 32-bit integer comparisons.

ρ_1 and ρ_3 have complete matching maps; moreover, ρ_1 has two valid matches (that we will call m_1^a and m_1^b, corresponding to

> `system sends a heartbeat to the base`
> (with score $-2K_{DROPPED} - 2K_{DIST}$)

and

> `base sends a heartbeat to the system`
> (with score $-2K_{DROPPED} - K_{DIST} - 11K_{INVERSION}$)

The valid match m_3 corresponding to ρ_3, on the other hand, has a matching score of 0 (the highest possible), since no terms are dropped and no inversions are present. As we will shortly see, the match m_2 corresponding to the application of ρ_2 to the outcome of applying ρ_1 and ρ_3 to s, has a matching score of $-K_{DIST} - 2K_{INVERSION}$.

Rules also have a scoring of their own, depending on their specificity:

$$\begin{aligned}
\mathsf{rscore}(\rho_1) &= 5K_{TERMS} + 3K_{TAGS} + 2K_{CONSTS} + K_{RULE}\\
\mathsf{rscore}(\rho_2) &= 3K_{TERMS} + 2K_{TAGS} + K_{CONSTS} + K_{RULE}\\
\mathsf{rscore}(\rho_3) &= 2K_{TERMS} + K_{TAGS} + K_{CONSTS} + K_{RULE}
\end{aligned}$$

The parsing algorithm will then proceed to try all the possible valid matches, in order of match scoring plus rule scoring. The process is then repeated recursively, as described by Algorithm 1. In the vast majority of cases, this strategy allows the parser to reach the optimal parse tree among the first few candidates. To account for the rarer cases, however, the algorithm carries out an exhaustive search for other solutions, subject in the actual implementation to a timeout limit needed to guarantee responsiveness.

In our case, the possible sequences of rule applications, in order of decreasing cumulative score, are

$$\begin{aligned}
\Sigma_1 &= \langle \rho_1[m_1^a], \rho_3[m_3], \rho_2[m_2] \rangle\\
\Sigma_2 &= \langle \rho_3[m_3], \rho_1[m_1^a], \rho_2[m_2] \rangle\\
\Sigma_3 &= \langle \rho_1[m_1^b], \rho_3[m_3], \rho_2[m_2] \rangle\\
\Sigma_4 &= \langle \rho_3[m_3], \rho_1[m_1^b], \rho_2[m_2] \rangle
\end{aligned}$$

where by $\rho[m]$ we denote the application of rule ρ with matching m. The parsing algorithm starts with the initial sa-structure $sa_0 = (s, \langle\rangle)$. The first step computes $sa_1 = \mathsf{apply}(\rho_1, m_1^a, (s, \langle\rangle))$, obtaining the new sa-structure

$$\begin{aligned}
sa_1 = (\;& \langle \texttt{the/DT @}_1\texttt{/ACT/EVT every/DT 10/JJ/CD second/NN/S} \rangle,\\
& \langle \langle \texttt{@}_1\texttt{,SEND system/... base/... heartbeat/...} \rangle \rangle\\
)\;&
\end{aligned}$$

where $@_i$ is the denotation for the internal identifiers used to link the various nodes in a parse tree. On the first member of sa_1 only ρ_3 is applicable; the next

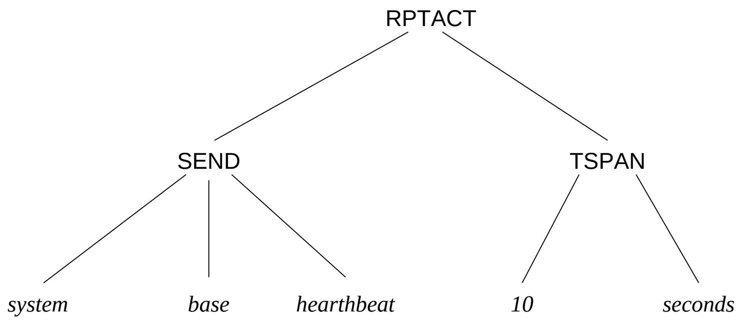


Figure 6: The parse tree for the requirement "The system shall send an heartbeat to the base every 10 seconds".

step thus produces

$$
\begin{array}{lll}
sa_2 & = (& \langle \texttt{the/DT } @_1\texttt{/ACT/EVT every/DT } @_2\texttt{/TSPAN}\rangle, \\
 & & \langle \quad \langle @_1\texttt{,SEND system/... base/... heartbeat/...}\rangle \\
 & & \quad\ \langle @_2\texttt{,TSPAN 10/... secs}\rangle \quad \rangle \\
 &) &
\end{array}
$$

Rule ρ_2 is now applicable, with match

$$\langle \underbrace{\{every\}}_{EVERY}, \underbrace{\{@_2\}}_{period}, \underbrace{\{@_1\}}_{action} \rangle$$

Thus producing

$$
\begin{array}{lll}
sa_3 & = (& \langle \texttt{the/DT } @_3\texttt{/ACT}\rangle, \\
 & & \langle \quad \langle @_1\texttt{,SEND system/... base/... heartbeat/...}\rangle \\
 & & \quad\ \langle @_2\texttt{,TSPAN 10/... secs}\rangle \quad \rangle \\
 & & \quad\ \langle @_3\texttt{,RPTACT } @_2\texttt{/TSPAN } @_1\texttt{/ACT/EVT}\rangle \quad \rangle \\
 &) &
\end{array}
$$

No further rules are applicable, and the parse tree thus obtained (that is shown in Figure 6) is assigned a score given by the sum of the matching scores of the matchings applied, plus the specificity scores of the rules applied, plus the score for the rests given by lscore(). In our case, the total score for the parse tree corresponding to Σ_1 is

$$
\begin{array}{ll}
-2K_{DROPPED} - 3K_{DIST} - 2K_{INVERSION} & \text{(matchings)} \\
+10K_{TERMS} + 6K_{TAGS} + 4K_{CONSTS} + 3K_{RULE} & \text{(rules)} \\
-2K_{REST} & \text{(rests)}
\end{array}
$$

The scores for the sequences $\Sigma_2 \ldots \Sigma_4$ are all worse than that for sequence Σ_1. The "optimal" parse tree for our sentence is thus the one given above. In other terms, the sentence

```
The system shall send a heartbeat to the
base every 10 seconds.
```

is transformed to

```
(RPTACT (TSPAN 10 secs)
        (SEND system base heartbeat)
)
```

that is more amenable to further processing. In concrete terms, the parse tree is represented as a set of tuples, where each tuple has a unique identifier that can be used to refer it. In our case, the parsing produces the following tuples:

```
1 SEND system/NN/SYS/IN/OUT/ELAB base/NN/IN/OUT heartbeat/NN/INF
2 TSPAN 10/JJ/CD SECS
3 RPTACT @₂/TSPAN @₁/ACT/EVT
```

where the first element of each tuple is a unique identifier, the second element is typically the *type* of the tuple, from the action part of the relevant rule, and the remaining arguments are either constants (as for "SECS" in tuple 2) or variable terms from the canonic form of the requirement (as for "base/NN/IN/OUT").

It should be noted that the parsing process implicitly performs a validation of the requirements with respect to syntactic quality [20]. For example, grammatical errors may be revealed by parsing failures, and the presence of potentially ambiguous words like "some" or "reasonably" (see for example [6, 21]) can be checked at parsing time by simply writing a few MAS rules to catch the offending keywords.

5 Model building & visualization

In the previous section we have discussed our first transformation (marked with ⓐ in Figure 4, leading from a NL requirements document to a forest of domain-based parse trees for the requirements. These parse trees correspond to the second element of the sa-structures returned by the parser for each requirement (see Section 4.3.4), and are encoded as a set of tuples. In addition to those tuples, the parser also produces a few other tuples maintaining statistical, linkage, and other administrative information for management purposes. All these tuples are inserted in the shared tuple space by the second transformation (marked with ⓑ).

We call such a set of related tuples a T-model. The T-model obtained from the parsing process constitutes the initial content of the shared tuple space, and is thus called the *initial* T-model. It abstracts out most of the syntactic and lexical variability of the original NL requirements, but is still very close to the source text. However, in the initial T-model requirements are encoded in a regular structure, hence they are more amenable to further processing.

Starting from the initial T-model, Circe builds more sophisticated T-models, focused on modeling, measuring, and validating the requirements. The process of building these models (the transformation marked c in Figure 4) and of displaying them to the user in a convenient fashion (transformations d and e) is the subject of this section.

5.1 Modelers

The task of building more T-models is carried out by a number of *modelers*. Each modeler, singularly or in cooperation with other, synthesizes new T-models from the contents of the shared tuple space. These T-models (encoded, as always, as a set of tuples) are inserted back in the shared tuple spaces, for other modelers to inspect or use in their own computation. Taken all together, CIRCE's modelers constitute a modular expert system, providing the intensional knowledge needed to successfully analyze software requirements.

In the following subsections we describe how modelers are specified and implemented, and how the global computation is organized by CIRCE.

5.1.1 Structure of a modeler

Formally, a modeler M is a triple:

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

where i is a set of tuple types, describing which kind of tuples M needs as input for its computation, o is also a set of tuple types, listing the types of tuples that M produces (i.e., the types that M uses to encode the T-model it computes), and $f : \mathcal{T} \to \mathcal{T}$ is a function from tuple spaces to tuple spaces, describing the specific computation. In practice, i and o constitute the *declaration*, or signature, of the modeler, while f constitutes its operational specification. A modeler must satisfy the three correctness properties below:

m1) In order to ensure non-retractability of requirements and of models,[9] f must be monotonic, i.e.,

$$\forall T \in \mathcal{T}, f(T) \supseteq T$$

m2) f must obey the input declaration in i, that is, the result must not be affected by extraneous tuples in the tuple space. Formally,

$$\forall T \in \mathcal{T}, \quad f(T) = f(\{t \in T \mid t = \langle id, type, \ldots \rangle \wedge type \in i\})$$

m3) f must obey the output declaration in o, that is, no new tuple should be added to the tuple space other than the ones specified in o. Formally,

$$\forall T \in \mathcal{T}, \quad \langle id, type, \ldots \rangle \in (f(T) \setminus T) \quad \Rightarrow \quad type \in o$$

[9] Unless, naturally, the text of the requirements themselves is changed.

For example, let us consider a modeler that synthesizes a T-model describing the interfaces between a software system and its environment. This modeler reads tuples referring to various forms of communications among entities in the domain (both components of the system described and environment entities), and produces other tuples — with a different type — that provide only a high-level description of the interfaces.

Formally, such a modeler is described by the triple

$$m_{iface} = \langle \{\text{SEND, RECEIVE}\}, \{\text{IIFACE, OIFACE}\}, f \rangle$$

where SEND and RECEIVE[10] are the types of the tuples describing communication actions, IIFACE is the tuple type for the T-model describing the input interfaces of the system, and OIFACE is the analogous type for the output interfaces T-model.

The f function could be described in logical terms as follows:

$$\begin{array}{rcl}
\text{iiface}(src, dst) & \leftarrow & \text{send}(src, dst, data) \wedge \neg\text{system}(src) \wedge \text{system}(dst). \\
\text{iiface}(src, dst) & \leftarrow & \text{receive}(dst, src, data) \wedge \neg\text{system}(src) \wedge \text{system}(dst). \\
\text{oiface}(src, dst) & \leftarrow & \text{send}(src, dst, data) \wedge \text{system}(src) \wedge \neg\text{system}(dst). \\
\text{oiface}(src, dst) & \leftarrow & \text{receive}(dst, src, data) \wedge \text{system}(src) \wedge \neg\text{system}(dst).
\end{array}$$

Intuitively, f is the function that, applied to a tuple space T, returns the whole of T (due to the monotonicity property) possibly supplemented with the IIFACE and OIFACE tuples corresponding to the SEND, RECEIVE, and SYSTEM tuples contained in T.

5.1.2 Modeler scheduling

As we noted above, modelers can cooperate to build more refined T-models. This means that the T-model produced by a modeler can be taken as input by other modelers to compute their own T-models.

Circe organizes the global computation by appropriately *scheduling* modelers according to their *dependencies*. Intuitively, a modeler m_1 depends on another modeler m_2 if m_1 needs in input some of the tuples produced by m_2. In these cases, m_2 should be called upon first to compute its T-model, and m_1 should be executed after the T-model of m_2 has been added to the tuple space. Since each modeler can depend on an arbitrary number of other modelers, as well as provide tuples to an arbitrary number of other modelers, the dependency structure is a graph.

Formally, let M be a set of modelers. The *dependency graph* of M is a graph $DG_M = (M, E)$ where E is the minimal set satisfying the following property:

$$\forall m_1 = \langle i_1, o_1, f_1 \rangle \in M, \forall m_2 = \langle i_2, o_2, f_2 \rangle \in M, i_1 \cap o_2 \neq \emptyset \quad \Rightarrow \quad (m1, m2) \in E$$

[10] Many other types of communications are considered in Circe; the corresponding tuple types are omitted here for brevity.

We disallow cyclic dependencies between modelers[11], thus DG_M is a directed acyclic graph, inducing a partial order on M.

Let now M^* be the set of all the modelers available in CIRCE, and Q be a set of tuple types describing which T-models are needed for a specific computation (e.g., to synthesize an abstract view). Some of the tuple types in Q may refer to tuples in the initial T-model, but others may refer to tuples produced by other modelers. The set of modelers H directly providing the tuples required by Q (the *head modelers* for Q) is given by

$$H = \{m = \langle i, o, f\rangle \in M^* \mid o \cap Q \neq \emptyset\}$$

The set of all the modelers which modelers in H depend on is given by the set A (the *active modelers* set) corresponding to the minimal fix point of the following definition:

$$A = H \cup \{m \in M^* \mid \exists m' \in A \text{ s.t. } (m', m) \in E^*\}$$

where $DG_{M^*} = (M^*, E^*)$. From DG_{M^*} we can extract a subgraph DG_A containing only the dependencies between modelers in the active set:

$$DG_A = (A, \{(m_1, m_2) \mid m_1 \in A \wedge m_2 \in A \wedge (m1, m2) \in E^*\})$$

Any complete order compatible with the partial order induced on A by DG_A is a valid *modeler scheduling* for the requested T-model. Let $m_1 < m_2 < \ldots < m_n$ be such an ordering, with $m_j = \langle i_j, o_j, f_j\rangle$, and let T_I be the initial T-model. Then the tuples requested by Q are computed as

$$\mathbb{R}(Q) = \{\langle id, type, \ldots\rangle \in f_n(f_{n-1}(\ldots(f_1(T_I)))) \mid type \in Q\}$$

In the current implementation, CIRCE computes a suitable linearization of the dependency graphs as described above, and executes all the modelers in a pipeline. This increases the throughput of the system by taking advantage of multiple processors if available.

5.1.3 Modelers' roles

While all the modelers are formally defined in the same way, in practice they are used for several different purposes, namely providing (i) abstraction, (ii) intensional knowledge, (iii) metrication, (iv) validation, and (v) visualization support.

The simplest class of modelers are the *abstraction modelers*. These implement a kind of forgetful functions on the tuples they consider: the resulting tuple space will contain (alongside with the original tuples) a *projection* of some subset of the input tuples. For example, the modeler m_{data} defined as

$$\begin{array}{rcl} m_{data} & = & \langle\{\text{SEND}, \text{RECEIVE}\}, \{\text{DATA}\}, \\ & & \quad \{\text{data}(d) \leftarrow \text{send}(x, y, d). \\ & & \quad\;\; \text{data}(d) \leftarrow \text{receive}(x, y, d).\ \}\ \rangle \end{array}$$

[11] This is a design decision, motivated by efficiency considerations. Fix-point computations can still be performed, but they must be confined inside a single modeler.

is an abstraction modeler, collecting all the data items that are exchanged between any two entities in our requirements, and listing them in DATA tuples. Often, abstraction modelers also perform some kind of *selection* alongside with the projection: the modeler m_{iface} that we discussed in the previous section is an example of an abstraction modeler that projects on SEND and RECEIVE tuples, and selects on SYSTEM tuples.

The second class comprises the *intensional modelers*. These are modelers that combine information extracted from various tuples, possibly describing unrelated areas of the system and of the domain, to obtain information that is in some sense *novel*. Obviously, a purely functional operator like our modelers cannot inject in the tuple space any new information that was not already implicitly in it. However, while the tuples obtained from the parsing process constitute the *extensional knowledge* that is available about the system and its domain, the intensional modelers provide the *intensional knowledge* that allows CIRCE to analyze the requirements in a deeper way.

Intensional modelers capture parts of the knowledge and experience of a requirements analyst: they relate different T-models, recognize standard patterns in the requirements and provide common modeling solutions for them, identify gray areas in need of further exploration, and in general provide most of the "intelligence" in CIRCE. For example, a modeler could inspect the SEND and RECEIVE tuples to discover common data paths between multiple agents and could propose implementing a bus among them rather than a set of point-to-point connections. In CIRCE, the set of modelers that provide a UML model for the system described in the NL requirements (see Section 7.7) belong to this class, together with many others.

The third class of modelers are the *metrication modelers*. Their purpose is to *measure* attributes of other models, as expressed by the tuples in the tuple space. This may include counting occurrences of some particular combination, as well as more complex measures. Modelers in this class collect, for example, function point counts for each subsystem (based on their I/O specification and on the internal computation described in the requirements), and provide estimates for related attributes. As for all the other modelers, the measures collected by metrication modelers constitute a T-model, and are thus put back in the tuple space, where they can be analyzed by other modelers. In CIRCE, these metrics are also used to build models of the requirements development process, and to identify potentially dangerous situations, in addition to simply displaying them to the user for manual inspection and analysis.

Identifying inconsistent, ambiguous, and incomplete requirements is the goal of modelers in the fourth class, the *validation modelers*. These modelers check that certain correctness properties on some T-model are satisfied, and emit new tuples with the results of such checks. The properties that are checked for may include simple intra-T-model checks (e.g., any data store in a data flow model whose value is never used is suspicious, and finding such a data store may hint at a missing requirement), or more complex inter-T-model checks (e.g., references to a particular instance from a class of similar entities in a behavior model are inconsistent with an entity-relationship model that explicitly states that only one instance of that particular class exists in our domain).

While many of the validation modelers in Circe checks for simple properties, the combination of many checks — and, more importantly, the guarantee that the checks are performed in a completely automated and repeatable way — is often of great help to the analyst. In our experience, over 90% of the errors made by novice requirements analysts entail the violation of some of the validation checks performed by the validation modelers.

Finally, the last class includes the *view support modelers*. These modelers, that are often akin to the abstraction modelers, take care of massaging the data in the tuple space in order to simplify their on-screen or printed presentation, as requested by the user. It is important to remark that a single T-model can be represented in many ways. For example, a finite state automaton can be presented either as a table or as a suitably annotated graph. View support modelers generate the tuples that encode the particular abstraction of the underlying T-model that is needed for the presentation, possibly omitting unneeded details or rearranging the tuples to ease the visualization task.

5.1.4 Architectural considerations

Using the same kind of software component — the modeler — and the same data structure — the T-model in a tuple space — for such broad range of tasks, offers many advantages.

In terms of architecture, it makes Circe simple to understand and maintain: the kernel of the system only takes care of scheduling the computation of the various modelers in response to the queries that come from the user. Moreover, it makes Circe extremely expandable: by writing a new modeler, that often amounts to a handful of lines of declarative code, the requirements engineer can add new models of the system or of its environment, can measure and analyze them (e.g., to check their validity), and can provide suitable representations.

In terms of expressive power, the uniform representation embodied by T-models allows the free combination of the various classes of modelers to obtain interesting results.

For example, in Circe abstraction and intensional modelers are used to build T-models of the system described by the NL requirements. Validation modelers run on these T-models to identify violations of the validation properties pertaining to those models. Metrication modelers are used on the output of the validation modelers in order to obtain a measure of the number of potential errors in the requirements. Intensional modelers build on top of these measures to obtain a T-model of the requirements development process. Other metrication modelers look at the process T-model, and provide estimates of such attributes as stability, maturity, etc. High-level validation modelers look at such measures, and if some of the values are found to be atypical, the user is warned, possibly through one or more view-support modeler.

While the example above may seem a little extreme, the free combination of abstraction, modeling, metrication, validation, and presentation is used throughout in Circe in a variety of contexts. It is not unreasonable to think at the intermixing of all these activities in CIRCE as the mirroring — albeit partial and approximative — of the various cognitive processes that happen simultaneously in an experienced requirements engineer's mind when faced with a new problem and a new set of requirements.

5.2 Model visualization

Automated analysis and validation, even when partial and approximate, is a useful addition to a requirements engineer's toolbox. However, this should not lead to an underestimation of the *cognitive* aspects, that are of the utmost importance in requirements engineering. As Dan Berry puts it [5], every attempt to put the human out of the loop in RE is fundamentally flawed.

In CIRCE, many T-models have one or more user-level *representations*. Most representations are graphical or textual, but a few present more complex interfaces, involving navigation among different items, dynamic generation of different perspectives, or exporting of representations toward other tools[12].

The generation of a representation for a T-model is a two-step process, involving (i) generation of an *abstract view* of a T-model, expressed by a set of drawing or layout commands in a *representation language*[13], and (ii) *translation* of the commands into *rendering instructions*. These two steps correspond to the transformations ⓓ and ⓔ in Figure 4, and are carried out by components called *projectors* and *translators*, respectively. In the current implementation of CIRCE, the actual rendering is performed by outputting an HTML page containing the representation. The primary user interface to CIRCE is thus a simple web browser, through which the user can input the various artifacts in the requirements document $\mathcal{D}$ (from Section 4.1) and inspect the representations for the various T-models generated from $\mathcal{D}$.

5.2.1 Structure of projectors and translators

Formally, a projector P is described as a quadruple

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

where i is a set of tuple types, describing which kind of tuples P needs as input, o is a label identifying the particular representation produced by P (this is the label the user sees in CIRCE's UI), and $f : \mathcal{T} \to \mathcal{L}_t$ is a function describing the specific computation. In other words, f maps a T-model into a description (in some representation language named t, with $\mathcal{L}_t$ denoting the set of all possible valid descriptions in t) that, when properly interpreted, will render the representation named after o. As for modelers, projectors must satisfy the following correctness property:

[12]For example, UML models generated by CIRCE can be exported through XMI and imported into other tools such as Rational Rose or ArgoUML.

[13]The current implementation of CIRCE has seven representations languages, covering simple and rich text output, tables, histograms, forms, and two kinds of customizable, interactive graphs. The latter are used for simple diagrams (e.g., finite state automata) as well as for complex representations (e.g., UML class diagrams).

p1) f must obey the input declaration in i, that is, the result must not be affect by extraneous tuples in the tuple space. Formally,

$$\forall T \in \mathcal{T}, \quad f(T) = f(\{t \in T \mid t = \langle id, type, \ldots \rangle \wedge type \in i\})$$

A translator T for a representation language has a simpler structure:

$$T = \langle t, f \rangle$$

where t is a label identifying the language (this is the same as the t in the definition of a projector), whereas $f : \mathcal{L} \rightarrow \mathcal{H}$ is the interpretation function that maps a description in the representation language t into the final rendered page (we used $\mathcal{H}$ here to indicate the domain of valid HTML pages).

Circe includes a number of translators for most common representation styles; these are listed in Table 1.

As an example of how projectors and translators work, let us consider a graphical representation for the interfaces T-model of Section 5.1.1, called simply "Interfaces". The projector responsible for that representation, p_{iface} is be described formally as

$$p_{iface} = \langle \{IIFACE, OIFACE\}, \text{“Debug/IFace”}, f, \mathsf{graph} \rangle$$

where f is the function expressed by the following algorithm:

Algorithm 6 The f function of the sample view generator v_{iface}.

output "TITLE", "Subsystem interfaces"
let T be the input T-model
for all iiface(x,y) in T **do**
  output "ARC", x, y
end for
for all oiface(x,y) in T **do**
  output "ARC", x, y
end for

Algorithm 6 generates a title directive and a number of edge declarations ("ARC") that define a graph. These commands are then passed to the graph translator, which in turn produces HTML code, with an embedded Java applet, to render the graph. The final result is the graph shown in Figure 7.

5.2.2 Interface considerations

Concrete views constitute the only means of interaction between the user and Circe. This uniformity has important benefits in terms of usability and consistency of the user interface.

When a user logs on, after the necessary authentication phase, a special "Control panel" view is automatically generated to bootstrap the system. This

Name	I/O	Description
datapanel	I/O	Provides an user interface based on HTML forms. A projector can specify a number of (l, t, n, v) quadruples where l is a label describing the data item, t is its type, n is its symbolic name, and v its default value. The datapanel translator produces a page with fields corresponding to the data items (e.g., a checkbox for a boolean value); the values set by the user are stored in the shared tuple space upon submission of the form.
debug	O	A utility translator; any command received is simply output *verbatim* to the resulting web page.
downloadable	O	The concrete view is stored in one or more files in a special per-user storage area. The translator produces a page containing links for downloading these files.
graph	I/O	The representation is a graph; the translator prepares a page with an embedded Java applet to allow the user to browse and possibly to edit the graph. The graph appearance and behavior (e.g., which layout algorithm to use) is completely parametric. For editable graphs, the user can manipulate the graph (e.g., by adding nodes or edges, or by changing display attributes for the graph). These manipulations are stored back in the tuple space, and can be used later on to automatically re-layout the same view according to user's preference. Figures 5 and 7 show simple read-only graphs; examples of more complex (and editable) graphs can be found in Figures 10 and 13.
histogram	O	The translator produces a classical histogram view on a set of numerical data. Labels for each data item can be optionally specified, as well as highlighting of particular values.
table	O	Provides a table-based representation, with rich text or arbitrary HTML content in each cell. Figure 5 shows two concrete views produced by table (in the upper-right corner).
text	O	Produces rich text output, with escapes to full HTML and to *verbatim* text to provide complete control over the appearance of the view. In Figure 5, the view on the lower-right corner is produced by the text translator.
textedit	I/O	Provides the functionality needed to edit simple text documents. This is used for Designations, Definitions, and Requirements documents, as well as for other administrative (e.g., configuration) files.

Table 1: Names and capabilities of the translators available in CIRCE.

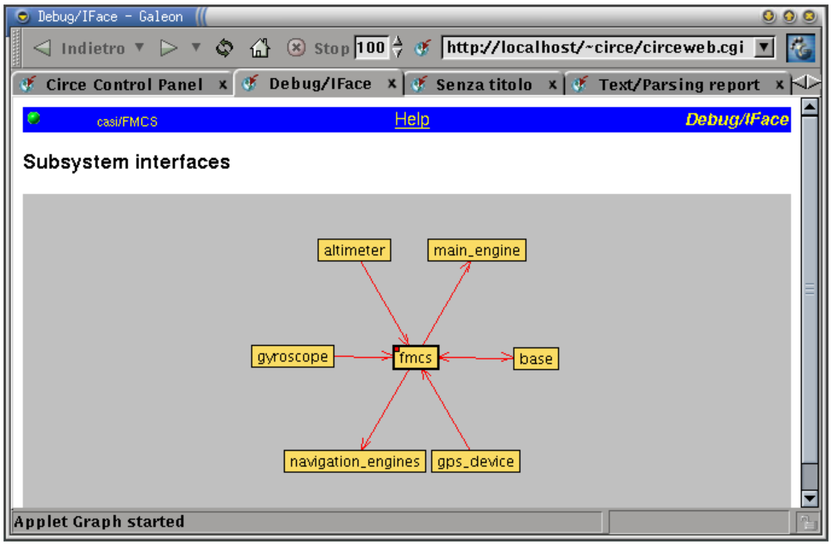


Figure 7: The final rendering of the view generated by v_{iface}.

datapanel-based view, that can be observed in the upper-left corner in Figure 5 (page 8), provides a list of all the projects and views available to that particular user. The user can freely choose any of them: in particular, a few textedit-based views provide a way to edit the requirements document itself, upon whose content all the other views are based. A number of other views also provide support for administration tasks, e.g., adding new users or establishing the access rights that users have on shared projects.

When the user selects a view v to display[14] from the "Control panel" window, the request is sent to CIRCE, which identifies the relevant projector as the one described by the quadruple $p^* = \langle i_p, v, f_p, t \rangle$. Thus, i_p indicates which T-models are needed to produce the view, whereas t indicates which translator $t^* = \langle t, f_t \rangle$ should be used to transform the abstract view generated by p^* into a concrete view. CIRCE then computes the needed T-models as described in Section 5.1.2), and instantiates the projector and the translator to compute the final result. In formal terms, this amounts to computing

$$\mathbb{H}(v) = f_t(f_p(\mathbb{R}(i_p)))$$

In practice, $\mathbb{H}(v)$ is a web page showing the requested concrete view labeled v, and this page is sent back to the user's browser for displaying. Having a web-based interface also has many pragmatically relevant advantages:

- CIRCE and its repositories can be accessed from everywhere (e.g., from the customer's site in the course of a discussion or interview) and no special

[14]The user can choose at any time whether new views should be opened in independent windows, or a pre-existing window should be recycled.

software needs to be installed on the client machine. This is particularly important in global software development [52, 14], especially during requirements negotiation;

- the user interfaces is standardized (e.g., the "back" command provided by the browser works as expected);
- interoperability with other applications is improved (e.g., it is possible to send a web page containing a representation of some model attachment to an e-mail discussing a certain design problem);
- it is easy to programmaticaly generate a static web site containing all the requirements-related documentation for a project, including requirements, definitions, designations, and representations for all the models that are deemed relevant, together with a full report of the validation checks passed.

Both the uniformity of the user interface (that is based on a single type of transaction — the request for a view) and the adherence to common web standards simplify life for the user, that can then concentrate on the challenging task of defining the right requirements without being distracted by a complicated user interface.

6 The Circe Native Meta-Model

In the previous two sections we have described the mechanisms that Circe uses to perform the chain of transformations, from NL requirements to concrete views on models extracted from the requirements. However, up to now we have said nothing on the specific MAS rules (see Section 4.3.1) used to parse the requirements, nor have we discussed which specific models are synthesized by our modelers (and especially by our intensional modelers, see Section 5.1.3).

These two elements together provide the reference framework that defines the way in which NL requirements are actually interpreted by Circe. This framework is called the Circe *Native Meta-Model*, or CNM for short. The CNM is not hard-coded in Circe: it is rather a result of the particular MAS rules, modelers, projectors, and translators used in the system[15].

We do not provide a full formal definition of the CNM in this work (the reader is referred to [3] for details), and in particular do not discuss document or process models, nor elaborate on metrication and validation. However, in the following we describe the basic underlying principles and most prominent features of the part of CNM that is directly concerned with the modeling of the system described by the requirements. We rely on the reader's intuition and knowledge of traditional software engineering modeling techniques to fill the gaps in the description.

[15] In fact, other meta-models have been implemented, for different purposes, by writing appropriate parsing rules and modelers. See, for example, [30], in which the argumentative structure of literary texts is analyzed and modeled instead.

In CNM, a reality is described along two different axes: a description of the *static aspects* of the various *entities*, and one of the *dynamic behavior* of a set of *agents*. While everything can be an entity (including agents, relationships, events, etc.), only those entities satisfying certain properties can be agents. Thus, agents are a subset of the entities in the system. In the following we examine in turn each of these aspects.

6.1 Entities and static aspects

Static aspects cover all the properties of a system that do not depend on time — not, at least, on a scale that is relevant in the applicative context. Thus, static aspects describe mostly structural properties of the system, e.g., the relationships among several entities or the attributes that characterize specific instances from a class of similar entities.

6.1.1 Entities, attributes, relationships

The tag `/ENTITY` is used to denote any physical or logical *entity* or class of entities that exist in the domain, and that are relevant for the requirements analysis. In fact, the tag `/ENTITY` is of such wide applicability that in most cases it can be omitted altogether[16] — its use is encouraged, however, for documentation purposes.

Entities can have *attributes* (tag: `/ATTR`), which in turn have *values* that can be static or mutable. For example, a physical entity "missile" can have a static attribute "length", whose value is some measure of the length of the missile, and a dynamic attribute "heading" giving its current direction. Less tangible entities can have attributes as well. For example, an event like the launch of a missile can have attributes "time of occurrence", "duration", or "cause".

Attributes are not necessarily independent from each other, although in most cases they will. For example, the attributes *weight* and *fuel load* of a missile are not independent: if the fuel tank of a missile is loaded full with fuel, its weight will increase accordingly.

Entities that are marked with the tag `/HASSTATE` have a distinguished enumerated attribute called the *state* or *mode* of the entity. Values of this enumeration, i.e. the possible modes in which an entity can be, are marked in $\mathcal{G}$ with the `/MODE` tag.

Whenever an agent observes an external entity, either directly or by observing some of its attributes, we assume that the agent will have an internal representation (a *reification* or hypostatisation) of the observed entity. It is the responsibility of the specifier to describe how this internal representation is kept in sync with the real state of the world. For example, some kind of sensor can be introduced, or a notification protocol can be used between active agents.

Entities can be related by describing the *relationships* that exist among them. Relationships have a name (tag: `/REL`) and cardinalities, as customary in E-

[16]Most MAS rules dealing with entities use the wildcard tag `/*` (from Section 4.5.3) rather than `/ENTITY` to support the simpler notation.

R models. A small set of well-known relationships, namely IS-A, PART-OF, CONNECTED-TO, and CONTROLS, are of special significance in the CNM. These relationships are used to build specific models of the system and of its domain, e.g. inheritance hierarchies and part-assembly diagrams, as well as for validation purposes.

Relationships are commonly static — that is, immutably embedded in the world — but can also be dynamic. In the former case, it is assumed that the implemented system will have a way of knowing the relationships, possibly by having them wired-in (for physical entities) or hard-coded (for software entities). A dynamic relationship is associated with some piece of information that reifies it. For example, a relationship between target names and their geographical coordinates for our missile system can be assumed to be static, e.g., embedded in some compiled-in table in the control software. Alternatively, we can declare it as a dynamic relationship. In this case, we can explicitly introduce the names-coordinates table and possibly describe how and when it is updated by the command center operator.

6.1.2 Instances selection

To identify particular instances from a class of similar entities, *selective expressions* can be used. A selective expression includes a linguistic *quantifier* (e.g., "all", "any", "some", etc.), a class of entities from which to select the desired instance, and a *selection predicate*. The latter can be (i) a simple predicate on the instance or on its attributes, (ii) a particular `/MODE`, or (iii) an expression denoting *navigation* from a given entity to the desired instance, according to the relationships described in the requirements. The latter case is of particular interest, since it involves the concept of *navigability* of a relationship. We define the *target* of x in a relationship R (written as $target_R(x)$) the set $\{y \mid (x, y) \in R\}$. A relationship R is said to be *1-navigable* if $\forall x, |target_R(x)| = 1$. To encourage precise writing, selective expressions using determinative quantifiers (e.g., the determinative article "the") can only be used when the existence and uniqueness of the selected instance can be assured, e.g. by using navigation on 1-navigable relationships. Other uses are also accepted, but flagged with a warning (suggesting the use of indeterminative quantifiers instead).

As an example, Table 2 lists some selective expression, together with an explanation of how they are interpreted in CNM. Other examples are provided in Section 7.

6.2 Agents and dynamic aspects

Agents in CNM can be active, reactive or passive, and can communicate among themselves either explicitly, or implicitly by observing some shared portion of their state. Active agents have their own "flow of control"; they can take autonomous actions and initiate chains of causal phenomena. Reactive agents are similar, in that they can react to events that they observe in the world, but they never initiate a causal chain on their own — they can only take part in such a

Expression	Definition
All ready missiles	$S = \{m \mid m \in Missiles \wedge status(m) = ready\}$
The engine of the missile	$S = \{e \mid e \in Engines \wedge$ $\exists R_1, \ldots, R_n, \exists x_1, \ldots, x_{n-1}$ s.t. $(m, x_1) \in R_1$, $(x_1, x_2) \in R_2, \ldots, (x_{n-1}, e) \in R_n\}$ where m is the given missile, and $R_1, \ldots, R_n$ are 1-navigable relationships
The LCC that controls the missile	$S = \{l \mid (l, m) \in CONTROLS\}$ where m is the given missile, and $\|S\| = 1$
All missiles whose fuel tank's level is greater than the minimum launch fuel level	$S = \{m \mid m \in Missiles \wedge$ $\exists R_1, \ldots, R_n, \exists x_1, \ldots, x_{n-1}$ s.t. $(m, x_1) \in R_1$, $(x_1, x_2) \in R_2, \ldots, (x_{n-1}, f) \in R_n \wedge$ $f \in FuelTanks \wedge$ $level(f) > MinimumLaunchFuelLevel\}$

Table 2: Some selective expression and their meaning in CNM.

chain as an intermediate of final link. Passive agents are a restricted form of agent: they can change their state in response to some external event, but they never propagate a causal chain by taking further actions.

6.2.1 Causality

The behavior of active and reactive agents is described through a set of Event-Condition-Action (or ECA for short) rules. Each rule specifies which action(s) should be executed when a certain event happens, if the given conditions are true.

Events, conditions, and actions can be expressed as uninterpreted designations from $\mathcal{G}$, but can also be given in more structured ways. Circe's model includes functional and timing aspects that can be used to express events and conditions. For example, a condition can be defined on the value of some data item, and an event can be fired when the data item changes its value. Actions include initiating data exchange, setting the value of data items, and other similar activities.

Events, conditions, and actions are not distinct sorts. The same linguistic expression can often serve more than a role, depending on the context in which it is found. For example, Table 3 shows how the same expression "the missile sends the logs data to the LCC" can express an event, a condition or an action, depending on the linguistic context surrounding it. To distinguish these different roles, in the following we will sometime decorate the name of an event/condition/action expression α with a superscript (e, c, or a) to indicate its exact nature in a particular context. Thus, we will use α^e for an event, α^c for a condition, and α^a for an action. In $\mathcal{G}$, terms denoting events and conditions are tagged with the `/EVT` tag, while terms denoting actions have the `/ACT` tag.

Formally, an ECA rule is a mixed sequence of events, conditions, and actions. For example, consider the sentence below:

Requirement	Role
As soon as the missile sends the logs data to the LCC, it shall …	Event
If the missile sends the logs data to the LCC, it shall …	Condition
Every two minutes, the missile sends the logs data to the LCC.	Action

Table 3: An example of how linguistic context influences the event, condition, or action role of an expression.

> When the FMCS enters operation mode, it turns on the main engine; if the engine is ignited correctly, then when ground speed exceeds 20 Km/h the FMCS enters flight mode.

The corresponding ECA rule is

$$\begin{aligned} r \quad = \langle \quad & \text{FMCS enters operation mode}^e, \\ & \text{FMCS turns on the main engine}^a, \\ & \text{engine is ignited correctly}^c, \\ & \text{ground speed exceeds 20 Km/h}^e, \\ & \text{FMCS enters flight mode}^a \qquad \rangle \end{aligned}$$

The intuitive meaning of ECA rules is that they describe a system that examines the various elements sequentially: checking that certain events have happened, waiting for other events, checking for conditions, and executing actions.

A more precise formulation is as follows. ECA rules that have an event as their first element are called *normal*; the event is said to be the *head event* for the rule. All other rules can be *normalized*:

1. if the rule includes at least an event element, and all the preceding elements are conditions, then the first event element is shifted to the first position, leaving all the other elements untouched (this operation does not change the semantics of the rule, under the assumption that checking a condition takes negligible time).

2. otherwise, it is intended that the activity described by the rule should be executed as often as possible. The rule is prefixed by a special system event Φ^e, that is assumed to happen with sufficiently high frequency. In other words, Φ^e simulates the timing of a polling loop.

The causal behavior of an agent is thus described by a set of normal ECA rules. At each computation step, the agent verifies which ones among the head events of its rules has happened since the preceding step, and *fires* the corresponding rules. Each element in a fired rule after the head event is then evaluated in turn. If it is an action, it is executed, and processing of the rule continues. If it is a condition, and the condition holds, processing of the rule continues, otherwise it is stopped. Finally, if it is an inner event, and the event has not yet happened since the previous step, execution of the rule is suspended, and will be resumed after the first occurrence of the event.

We assume that rule firing is non-blocking: several rules can be fired at once, and more rules can be fired while the execution of fired rules is still in progress or suspended waiting for an inner event.

The causality model in CNM is not as general and flexible as those provided by process algebras or Petri nets. However, its main virtue lies in being close to the intuitive meaning of the natural language counterparts of its ECA rules, namely of NL requirements. Moreover, the basic principles behind causality in CNM are very similar to those in the computational model of Abstract State Machines [9], of which our model is a superset [29].

6.2.2 Actions

The number and variety of actions that an agent can perform is rather limited by the inherently limited capabilities of software artifacts. In particular, in CNM agents can perform input and output, store and retrieve values from a storage area, and perform computations on these values. Of course, these are all low-level actions: depending on the domain, higher-level actions can be defined either as designations in $\mathcal{G}$ or via definitions in $\mathcal{F}$. Agents that can perform input are tagged with the `/IN` tag, while those that can perform output are tagged with `/OUT`. Thus, *sensors* are typically tagged `/OUT` only, *actuators* are tagged `/IN` only, and interacting agents are tagged `/IN/OUT`.

Several variations of input and output primitives are provided, including sender-initiated synchronous or asynchronous communications, receiver-initiated queries, and asynchronous interrupts. Implicit communications (e.g., when an agent predicates on the value of an attribute of some other entity) are also allowed. Special linguistic forms for interacting with human users and common devices and subsystems (e.g., printing, reading from a keyboard, interacting with a DBMS, etc.) are defined through MAS rules, and mapped to the more primitive forms of communication described above.

Values received or sent through I/O operations are stored in *data stores* named after the kind of data. These names are declared in $\mathcal{G}$ with the `/INF` tag, denoting terms that have some information content. In addition, *reification* is applied to implicit I/O operations, as described in Section 6.1, so that observed entities have an associated data store, corresponding to the model of the external entity that the observing agent maintains. Often, data stores have attributes (either defined explicitly or derived from those of reified entities), that in this case corresponds to fields of a structured value.

Arbitrary computations on the values contained in data stores and on their attributes can be described in the requirements, provided that the agent performing them has the `/ELAB` tag. CNM does not provide any means to specify a particular algorithm or formula for the computation: this is better expressed via mathematical expressions or pseudo code supplementing the requirements. However, special linguistic forms for a number of simple operations on common data types are available through MAS rules. Some of these are given in Table 4.

Beside performing I/O and computations, agents with computational capabilities (`/ELAB`) that in addition have a state (`/HASSTATE`) can also change their

Expression	Definition
appends x to y	$y' = y \bigoplus x$
combines $x_1 \ldots x_n$ giving y	$y' = f(x_1, \ldots, x_n)$
computes y from $x_1 \ldots x_n$	$y' = f(x_1, \ldots, x_n)$
copies x to y	$y' = x$
deletes x from y	$y' = y \bigominus x$
extracts x from y	$y' = y \bigominus \mathit{first}(y) \ \wedge \ x' = \mathit{first}(y)$
updates y using x	$y' = f(y, x)$
Legend	
x'	new value of x after the computation
f	unspecified function
$y \bigoplus x$	the collection y, with the element x added
$y \bigominus x$	the collection y, with the element x removed
$\mathit{first}(y)$	a distinguished element of the collection y

Table 4: Some of the special linguistics forms to express various kinds of computations.

state at will.

Finally, both arbitrary functions (tagged with `/FUN`) of an agent and completely abstract actions (tagged with `/ACT`) can be defined in the glossary. Comments can be added in the glossary to provide informal descriptions of functions and actions.

6.2.3 Events and conditions

As discussed above, the role of event or condition for a linguistic expression does not depend on the expression itself, but rather on the context surrounding it. In the following, we use the terms "condition" or "event" interchangeably, with the understanding that a condition can be true or false at any time instant, and that an event is considered to have occurred when the corresponding condition changes from false to true.

In CNM it is assumed that the actions performed by some agents are observable by other agents. Thus, all the linguistic forms describing actions that we have seen above can also be used as events[17]. In addition to that, and to abstract events declared with `/EVT` in $\mathcal{G}$, a number of other events can be expressed.

Monadic, diadic, and relational operators on values can be defined with the `/UNOP`, `/BINOP` and `/RELOP` tags. For the latter, a number of common relational operators are pre-defined (e.g., equals-to, greater-than, etc.).

Receiving some information via an I/O operation initiated by the partner is also an event, as is changing the current state of the agent.

[17]With the minor exception that abstract actions defined with `/ACT` in the glossary are not automatically considered events, unless the `/EVT` tag is also given. This caters for the definition of purely internal (and unobservable) actions, should the need arise.

Expression	Definition
Events	
x/EVT after y/PIT	an occurrence of x after time y
x/EVT before y/PIT	an occurrence of x before time y
x/EVT during y/TINTERV	an occurrence of x inside the interval y
x/EVT for y/TSPAN	an occurrence of x lasting at least y
x/PIT after y/EVT	a cyclic occurrence of x after an occurrence of y
x/PIT before y/EVT	a cyclic occurrence of x before[18] an occurrence of y
x/TSPAN since y/EVT	the event that occurs exactly x time after each occurrence of y
Constraints	
x/ACT y/PIT	x must be started at time y
x/ACT before y/PIT	x must be completed before time y
x/ACT every y/TSPAN	x is executed cyclically every y
x/ACT every y/TSPAN since z/PIT	x is executed cyclically every y, with the first execution scheduled at time z
x/ACT while y/TINTERV	x is executed while the interval y lasts

Table 5: Time-related expressions in CNM.

6.2.4 Time aspects

CNM defines three main time-related concepts: the time instant or point-in-time (`/PIT`), the duration or time span (`/TSPAN`), and the time interval (`/TINTERV`).

`/PIT` terms describe fixed (e.g., midnight of January 1, 2000) or modular-cyclic (e.g., everyday at noon) time instants. `/TSPAN`s express the difference between two instants (e.g., 30 minutes), while `/TINTERV` terms denote fixed intervals between two given instants (e.g., the 3 hours between 3pm and 6pm of January 1, 2000).

These temporal expressions can be combined with other events to denote time-delayed events, or with actions to describe constraints on the execution time of the actions. Table 5 shows some of the linguistic forms used with time-related terms.

In addition, time terms can be constructed from common units (e.g., second, minute, hour, day, etc.) and from other terms (e.g., "from x/PIT to y/PIT" is a `/TSPAN`).

7 A complete example

In this section we come back to our example regarding the Fictitious Missile Control System. Wearing the hat of the requirements engineers in charge of the FMCS, we show how Circe can be used in practice to write requirements that are understandable to the customer, yet can be the subject to rigorous validation checks. At the same time, we show how different representations of the requirements, automatically synthesized by CIRCE, help us in better visualizing and analyzing the FMCS requirements.

7.1 Problem statement

Informally, the FMCS problem can be stated as follows.

We want to develop a control system to direct the course of a missile. The on-board control system waits for an activation command coming from its launch control center (or "base"); after a certain security protocol has been completed, the system commands ignition and takes control of the flight until a pre-established cruise altitude is reached. Being generally peaceful people, in this example we do not investigate what happens after that stage. During flight, the FCMS monitors a number of parameters, and transmits them to its base for telemetry purposes. Moreover, a number of timing constraints are in place, and need to be described in the requirements.

7.2 A first sketch

At first, we write a few statements describing the domain in which our FMCS will have to operate:

Requirements

```
A launch control center manages a number of missiles.

Each missile is controlled by an on-board FMCS unit.
```

Of course, we have to declare which designations we are using in these statements.

In our case, we already have a number of designations, and take the opportunity to also declare a few synonyms for them.

Designations

```
base/ENTITY: launch control center.
missile/ENTITY.
FMCS/ENTITY: missile control system, control system, system,
FMCS unit.
manage/REL.
```

By declaring these designations, we ensure that all the parties involved in the requirements development process have recognized that a certain domain element exists, and agree on its naming. In addition, we clearly state a few properties on the nature and capabilities of the entity so designated.

Notice that we do not provide a designation for "controlled by", as this particular relationship is pre-defined in system glossaries in CIRCE (together with a handful of others).

If at this stage we look at the Parsing Report view provided by CIRCE, we can see that "on-board" has been treated as a rest (using the terminology established in Section 4.3.6), that is, it has been ignored. This is not surprising, since CIRCE does not have a model for the physical placement of objects, and the fact that the FMCS unit is on-board the missile rather than in any other location is largely irrelevant to our analysis (of course, it *is* relevant for other purposes, e.g. ensuring fast and safe communication channels between the FMCS and the various sensors/activators that are on the missile). No other parsing errors are shown, confirming the syntactic correctness of our requirements.

The relationships that we have declared will turn out to be useful in the following to exactly identify, among the possibly many missiles, FMCSs and control centers, which one actually participates in specific interactions.

We can now turn our attention to the ignition procedures. From an abstract point of view, we could ignore the details of the protocol and just write

Requirements

```
When launch is confirmed, the FMCS shall ...
```

This approach would entail simply declaring a designation for "launch is confirmed":

Designations

```
launch is confirmed/EVT.
```

However, in this particular problem the exact confirmation protocol is of special importance (for security purposes), so we decide to write it down in detail in the requirements specification. Investigating the matter with the customer, we discover that a launch is confirmed only if the FMCS receives a launch command, that has a specific (random, long) key as parameter, followed by a distinguished launch confirmation command having the same key. A launch cancellation command at any stage would interrupt the procedure. This protocol can be expressed as follows:

Requirements

```
The FMCS is initially idle.

When the FMCS receives a launch command from its base, it
enters confirmation waiting mode.

If

  • the FMCS is in confirmation waiting mode,
  • it receives a launch confirmation command from its base,
    and
  • the token of the launch confirmation command is equal to
    the token of the launch command,

then the FMCS enters operation mode.

When the FMCS receives a launch cancellation command from its
base, it becomes idle.
```

This specification introduces a number of interesting issues. First, we are now assuming that the FMCS can be in several different *modes*. We must declare this fact in the designation for FMCS, and also introduce designations for the various modes:

Designations
```
FMCS/ENTITY/HASMODE: ...
idle/MODE.
confirmation waiting/MODE: confirmation waiting mode.
operation/MODE: operation mode.
```

Second, we are assuming that the FMCS and its base are communicating; in particular, we are assuming that the FMCS receives commands from its base. We have thus to add these capabilities to the designations for FMCS and base:

Designations
```
FMCS/ENTITY/HASMODE/IN: ...
base/ENTITY/OUT: ...
```

Moreover, we have to introduce designations for the various commands that are sent by the base, and for the "token" parameter of some commands:

Designations
```
launch command/INF.
LCC/INF: launch confirmation command.
LXC/INF: launch cancellation command.
token/ATTR.
```

Finally, we must declare that the FMCS is capable of performing computations, taking decisions, and autonomously change its state; all these capabilities require computational power, that is expressed by the tag `/ELAB`:

Designations
```
FMCS/ENTITY/HASMODE/IN/ELAB: ...
```

The relational operator "is equal to" is pre-defined in CIRCE, so we do not need to explicitly introduce it in the designations. Had we wished to use some special-purpose operator, e.g. checking a cryptographic signature on the tokens, we could have introduced such operators explicitly in the glossary, with tags `/UNOP`, `/BINOP`, or `/RELOP` as needed.

7.3 Initial models

The few requirements that we have written are already enough to start observing some model generated by CIRCE. Figure 8 shows a state transition diagram representing the various states in which the FMCS can be, together with the respective transitions. As can be observed, our requirements specify that upon receiving a launch cancellation command *at any stage* — even after entering the operation state, and possibly having ignited the engine —, the system returns in its idle state. This is suspicious, and may or may not be what the customer had in mind when providing the informal description of the launch protocol. In any

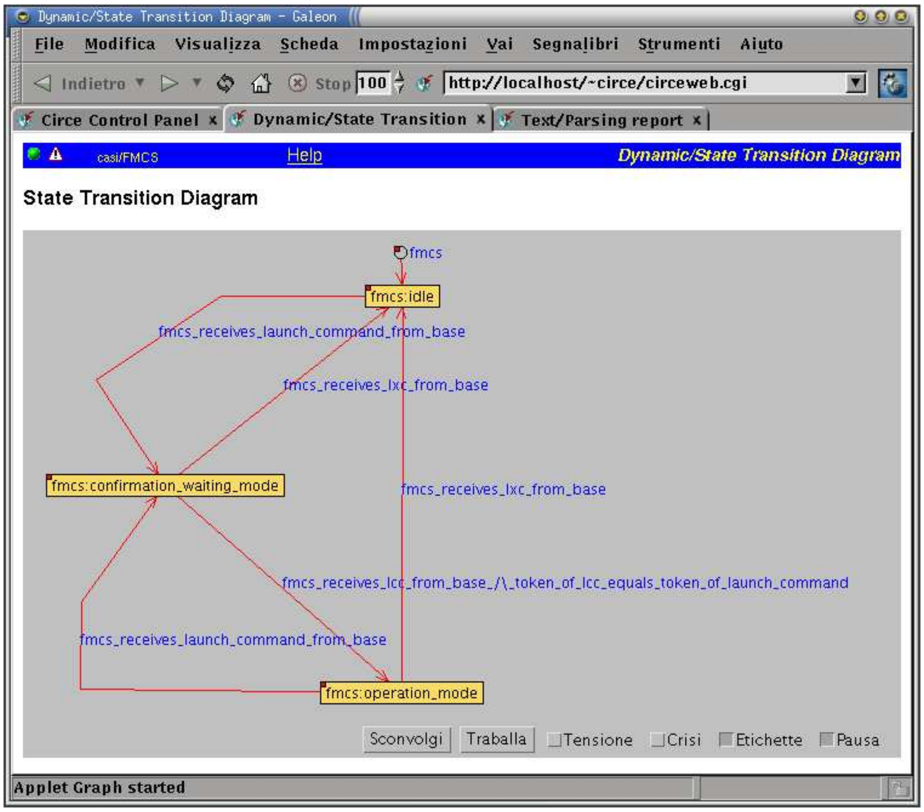


Figure 8: A state transition diagram synthesized from our initial requirements.

case, the diagram constitutes a good basis for discussion, exposing a potentially dangerous misunderstanding.

We decide to return again to the customer with our diagram, and come back after having discovered that launch cancellation commands are to be accepted *only* while in confirmation waiting mode (and silently ignored in other cases), and that launch commands are to be accepted only while in idle mode. We rewrite thus the relevant requirements as

---

Requirements

> `When the FMCS receives a launch command from its base, if it is idle, the FMCS enters confirmation waiting mode.`
>
> `When the FMCS receives a launch cancellation command from its base, if it is in confirmation waiting mode, it becomes idle.`

7.4 After take off

The next step in our description is to write down the requirements for the operational phase of the FMCS. The first thing to do when launch is confirmed, is to turn on the main engine:

Requirements

> `When the FMCS enters operation mode, it turns on the main engine.`

Again, this requirement introduces a number of new elements in our domain. First, "main engine" appears as a rest, as it is not known yet. We have to introduce a designation for it:

Designations

> `main engine/ENTITY/IN.`

---

to signify that the main engine is a physical entity that is also capable of receiving information (in our case, commands). It is also useful to record the relationship between the main engine and other elements in our domain:

Requirements

> `A main engine is part of a missile.`

Second, the concept of "turning thing on" (or off) is not primitive in the CNM, so "turns on" is reported as a rest. We have to introduce a definition for it:

Definitions

> `TURN x ON` $\mapsto$ `SEND AN ACTIVATION COMMAND TO x`
>
> `TURN x OFF` $\mapsto$ `SEND A DEACTIVATION COMMAND TO x`

---

where "activation command" and "deactivation command" are new designations:

Designations

```
activation command/INF.

deactivation command/INF.
```

Then, we have to specify what other operations are conducted during take-off, and when the FMCS leaves this state. Moreover, a timeout is used to detect any malfunctioning in the engine, and bring the missile to a safe state in that case.

Requirements

```
While the FMCS is in operation mode:

  • every 100 milliseconds, it reads the height from the
    altimeter, and
  • if the height is greater than 30 meters, it enters cruise
    mode.

If the FMCS is still in operation mode 5 seconds after it
entered that mode

  • the FMCS turns off the main engine, and
  • it enters idle mode.
```

For brevity, we omit from now on "plain" designations introduced by new requirements (in this case: `height/INF`, `cruise mode/MODE`, `altimeter/OUT`, etc.), showing only the most significant ones.

7.5 Cruise activities

During the cruise phase, the FMCS should continuously update its position and provide appropriate navigation command to the engines in order to reach its appointed target. To avoid unnecessary wear on the engines, the commands should be sent only if they prescribe a "significant" deviation from the current course (otherwise, the commands are discarded and the current route is kept — deviations will accumulate until they become significant). The following requirements provide a precise statement of these activities.

Requirements

```
During flight, every 20 milliseconds:

  • the FMCS reads the height from the altimeter,
  • it reads the heading from the gyroscope, and
  • it computes appropriate navigation commands, based on the
    target position.

During flight, if the navigation commands are significant, the
FMCS sends these commands to the navigation engines.
```

If we simply submit this requirement to CIRCE, the parser will warn us that `during flight` and `appropriate` cannot be parsed correctly (they are rests). Indeed, usage of "appropriate" and similar vague adjectives has long been identified as one of the most common pitfalls in natural language requirements. If we really want to use it in this context, we have to explicitly declare that "appropriate" is actually devoid of meaning (in a sense, we state that we are using it for purely euphonic reasons). In a similar vein, we have to declare (possibly after having consulted the customer) that by "flight" we intend that part of the operations that take place *after* the missile has been launched and reached the minimum quote of 30 meters. These declarations must be made by writing:

Definitions

```
DURING FLIGHT  ↦  WHILE THE FMCS IS IN CRUISE MODE
APPROPRIATE x  ↦  x
```

Notice how the linguistic analysis performed by CIRCE has forced us to explicitly declare a precise interpretation for our requirements.

A different mechanism is at work with the "significant" adjective. In this case, we decide to stay abstract from the details of which commands are significant, by simply writing

Designations

```
significant/UNOP.
```

By using a designation instead of a definition or a more complex requirement, we state that the meaning of "significant" is not given in our document: we assume it to be a basic term. Of course, a definition will be needed at the implementation phase, but it will be provided by different means (e.g., a mathematical formula or a pseudo-code algorithm).

Finally, the FMCS has to provide telemetry data to its base, by periodically reporting the location of the missile. In a realistic application, much more data would be logged, but let us exploit the "F" in "FMCS" by considering only the missile position, heading, and height to be relevant:

Requirements

```
During flight, the FMCS logs position, height, and heading
every 2 seconds.
```

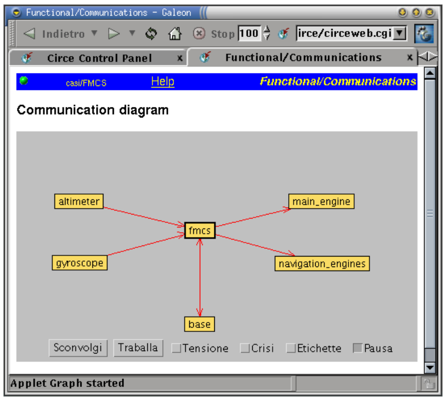
The concept of "logging something" is not part of the CNM, so we must provide a definition for it

Definitions

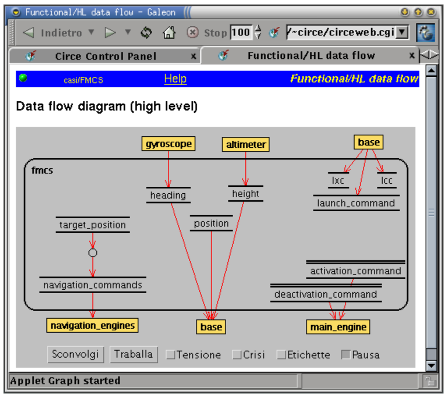
```
FMCS LOGS x/INF  ↦  FMCS SENDS x TO BASE
```

Using a definition for this kind of logging may seem baroque at first, but it helps to clearly distinguish the different purposes of the various communications[19], and has additional advantages in facilitating the evolution of the

[19] In fact, we have used the same technique when we used a definition to establish the meaning of "turning things on/off".



(a)



(b)

Figure 9: (a) The Communications model for the sample requirements. (b) The High-Level Data Flow model for the same requirements.

requirements. For example, let us suppose that at some future time it is decided to introduce a form of security by encrypting all data before they are transmitted to the base. This can be obtained in a general way, and without changing the requirements, by extending the definition above as follows:

Definitions

```
FMCS LOGS x/INF  ↦  FMCS CRYPTS x AND FMCS SENDS CRYPTOGRAM
OF x TO BASE
x/ELAB CRYPTS y/INF  ↦  x COMPUTES CRYPTOGRAM OF y FROM y
```

In a real case, we could continue detailing the behavior of the FMCS for several pages, but for our illustrative purpose the requirements written so far are sufficient.

7.6 Validation and further models

It is now time to check the correctness of the requirements (together with the designations and definitions) that we have written so far. CIRCE helps in this stage in two ways:

1. by verifying that the requirements satisfy a number of intra-model and inter-model *consistency properties*, defined and checked by CIRCE's validation modelers (see Section 5.1.3), and

2. by providing a number of different *views* on the requirements, that allow both the requirements engineer and the customer to visualize the requirements and check that they faithfully represent the customer's intentions.

As an example of this second kind of validation, Figure 9(a) shows a view depicting the data flows between the various subsystems (agents) in our domain. We can see (and show to the customer) that the FMCS reads data from the altimeter and gyroscope, and sends commands to the main and navigation engines. Moreover, there is a two-way communication channel between the FMCS and its base (this connection carries launch-time commands and telemetry data). This is actually what we would have expected; hence, we can rest assured that this particular aspect of our requirements is correct.

The first kind of validation is performed by looking at the views synthesized from data produced by the validation modelers. A view called *Pandora* (named after the Greek myth about Pandora's box, a chest containing all the world's evils along with a single blessing: Hope) presents a summary of the violations of all the validation properties. In our case, we can find — among others — the following warning (taken verbatim from the Pandora view):

> **Undefined information**
>
> The values of *target_position* and *position* are never modified by the system.
>
> *A variable that is never modified is not a variable. If it is a constant, it should be declared with the /BLTIN tag in the glossary. Otherwise, you should add some requirement explaining how the value of the variable is obtained.*
>
> <u>Relevant views</u>: Functional/HL data flow, Functional/LL data flow.

By looking at the high-level data flow diagram in Figure 9(b), as suggested, the origin of the problem is immediately clear. We never said in our requirements how the FMCS determines which is the current position of the missile (that is to be sent to the base, together with other telemetry data), nor how the target is assigned to the missile. To solve these problems, we can — after having consulted the customer on the details — add the following requirements:

Requirements

```
The Launch Control Center sets the target position for the
FMCS of its missiles.

During flight, the FMCS reads the current position from its
GPS device every 5 seconds.
```

Unfortunately, as pointed out again by Pandora, this additions introduce a new problem. In fact, the selective expression "the FMCS of its missiles" is interpreted as "the FMCS that controls the missile that is managed by the Launch Control Center", as another view informs us. But the selection is underspecified, since a single Launch Control Center *manages* a number of missiles (see Section 7.2). Further investigation is needed to decide whether the Control Center sets the target position of *all* the missiles it manages at once, before launch, or if the target of *any* missile can be changed at will in an asynchronous fashion, and possibly even after take-off. We choose at this stage to leave the requirement underspecified: Circe will nevertheless record the problem, and consistently and repeatably present it as part of future validation. This will not prevent us from continuing adding and revising requirements as needed: in effect, we are simply tolerating this problem, postponing its resolution to a later stage, as has been often suggested [4, 24, 26].

Similar defects are also found regarding the relationships between the various components of our missile. For example, Circe warns that no relationship has been declared between the GPS device and the FMCS: a relationship is needed to ensure that the selective expression "its GPS device" in the second requirement we added above is correct. Indeed, good practice dictates that we should explicitly list the components of a missile, as we did for "main engine" in Section 7.4. This can be obtained by writing

Requirements

```
A missile has as components:
  • a main engine,
  • a set of navigation engines,
  • a gyroscope,
  • an altimeter, and
  • a GPS device.
```

In this way, such references as "the FMCS reads the current position from its GPS device" will be correctly interpreted as referring to "the GPS device that is part of the missile that is controlled by the FMCS". Moreover, we have enriched our Entity-Relationship model, shown in Figure 10, that provides further opportunities for validation with the customer.

Dynamic aspects of the system described by our requirements can be inspected as well. As an example, Figure 11 shows the chain of dependencies leading to the action "FMCS enters cruise mode".

In such a simple case, the same information could have been obtained by direct inspection of the requirements. In larger specifications, however, the information needed is often scattered all over the document: collecting it all — and making sure that nothing is overlooked — may be a difficult task. In our case, Circe assures us that the *only* way for the FMCS to enter cruise mode is by reading a height greater than 30 meters from the altimeter while in operation mode.

7.7 UML and code generation

Our requirements for the FMCS up to this point, albeit of course far from complete, are already enough to start thinking about a design and a prototype for the final implementation. Circe comes to help in this respect by providing views for UML Use Case Diagram, Class Diagrams, Sequence Diagrams, Collaboration Diagrams, and State Transitions Diagrams [49, 2]. In addition, Circe can generate pseudo-code for the system described by the requirements, in the form of Abstract State Machines [9, 29] specifying the behavior of each agent.

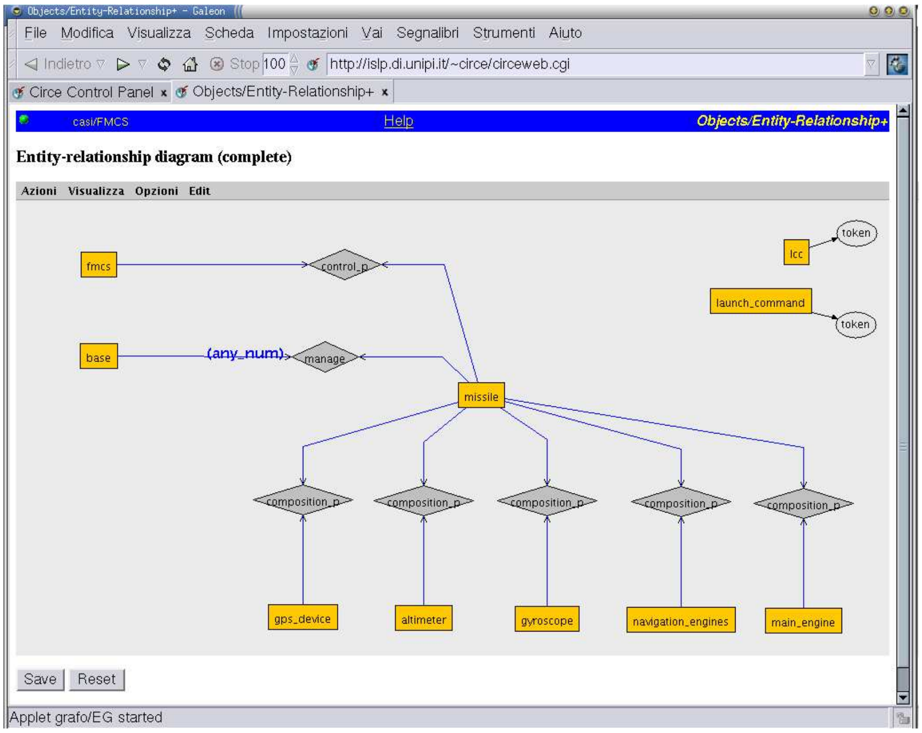


Figure 10: The Entity-Relationship model for the sample requirements.

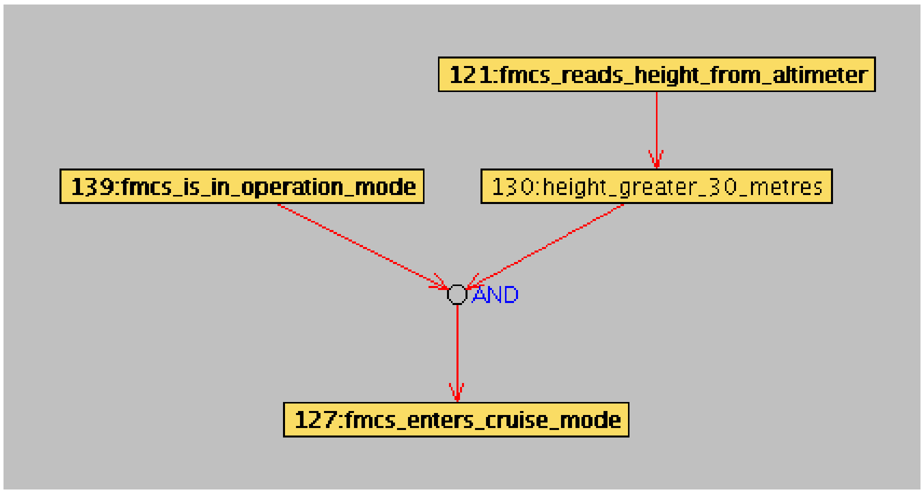
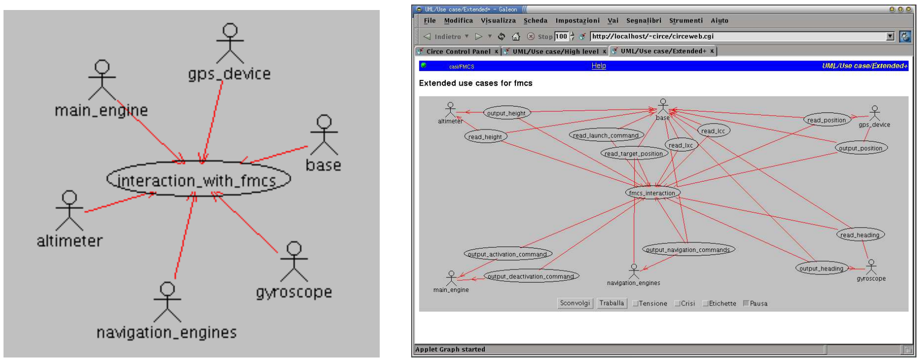


Figure 11: The dependency view for "FMCS enters cruise mode". The numbers prefixed to the entries are keys used to distinguish multiple instances of a fragment in the requirements.



(a) (b)

Figure 12: High-level (a) and detailed (b) use cases from the FMCS requirements.

As an example, Figure 12 shows the high-level and detailed view of the use cases identified by CIRCE in our requirements, whereas Figure 13 shows the UML Class Diagram generated from the requirements.

If we are satisfied with the resulting models, we can directly export them through XMI [42] to other tools (e.g., Rational Rose) and continue the detailed design activity in a more specific environment. Otherwise, we can continue refining our requirements by directly changing them or by refining our definitions, until a satisfactory model has been obtained.

But let us say that we are not interested in UML models after all, and will rather prefer a more state-oriented view of our requirements. In this case, we can ask CIRCE to generate pseudo-code for our system.

Algorithm 7 shows the resulting ASM rules for the FMCS. Of course, this code is very abstract, and heavily relies on to-be-defined macros (in small caps) to specify the details of the various operations. However, this representation has several advantages, including the fact that it is easily understandable to the developers.

Summarizing, in the course of our presentation of the requirements for the FMCS, we have shown how requirements can be defined in an incremental style, and how CIRCE supports the requirements analysis activity by encouraging the use of a definite writing style, by reporting syntactic and semantic (via model validation) errors, and by presenting a number of specifically focused views on various aspects of the requirements. Moreover, CIRCE does not abandon the requirements engineer at the end of the process, facilitating instead the transition to the detailed design and prototyping activity by providing UML- and ASM-based views on the requirements that can be exported to other tools.

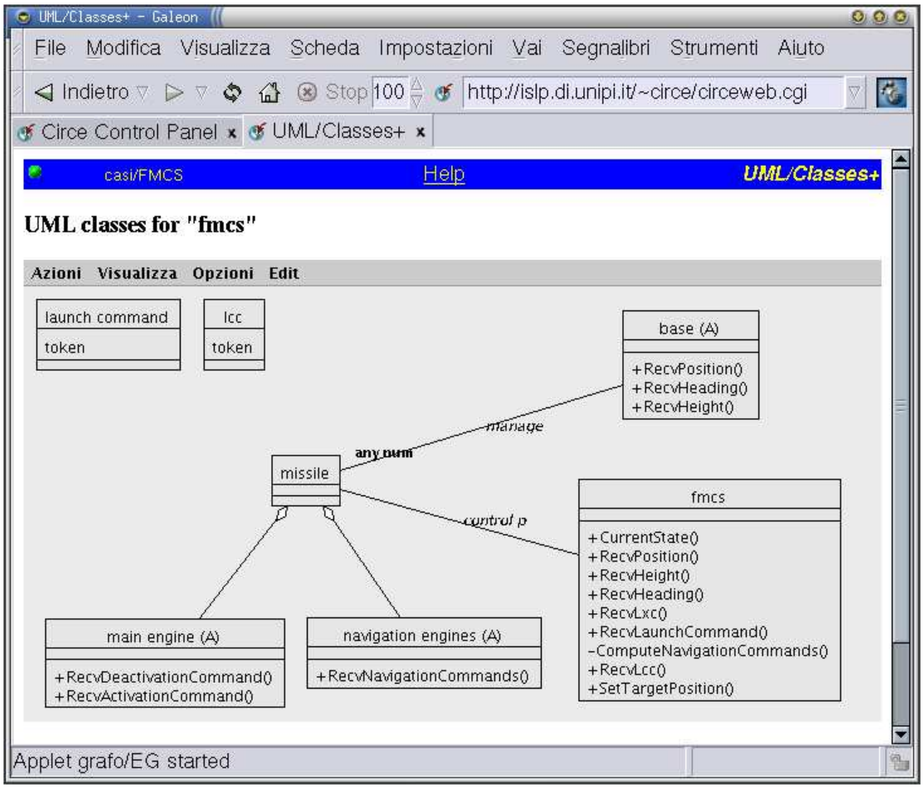


Figure 13: UML Class Diagram synthesized by Circe from the FMCS requirements.

Algorithm 7 ASM rules for the FMCS synthesized from the requirements.

Machine FMCS $\equiv$

if $\textsc{Recv}(Base, launchCommand)^e$ **and** $State(self) = Idle$ **then**
  $State(self) := ConfirmationWaitingMode$
end if
if $State(self) = ConfirmationWaitingMode$ **and** $\textsc{Recv}(Base, lcc)$ **and** $Equals(Token(lcc), Token(launchCommand))$ **then**
  $State(self) := OperationMode$
end if
if $\textsc{Recv}(Base, lxc)^e$ **and** $State(self) = ConfirmationWaitingMode$ **then**
  $State(self) := Idle$
end if
if $State(self) := OperationMode^e$ **then**
  $\textsc{Send}(MainEngine, activation\_command)$
end if
if $State(self) = OperationMode$ **and** $Greater(height, 30\_METERS)$ **then**
  $State(self) := CruiseMode$
end if
if $\textsc{5SecondsAfterFMCSEntersOperationMode}^e$ **and** $State(self) = OperationMode$ **then**
  $\textsc{Send}(MainEngine, deactivation\_command)$
  $State(self) := Idle$
end if
if $\textsc{20Milliseconds}^e$ **and** $State(self) = CruiseMode$ **then**
  $\textsc{Recv}(Altimeter, height)$
  $\textsc{Recv}(Gyroscope, heading)$
  $navigationCommands := f(targetPosition)$
  step
  $\textsc{Send}(NavigationEngines, navigationCommands)$
end if
if $\textsc{2Seconds}^e$ **and** $State(self) = CruiseMode$ **then**
  $\textsc{Send}(Base, position, height, heading)$
end if

8 Related work

Both the necessity of using natural language in requirements and the usefulness of building models based on those requirements have long been recognized in the literature. In [15], Al Davis states among his principles of Software Engineering: "Increase, never substitute, natural language". Five years later, in his 2000 review and agenda paper [51], Axel van Lamsweerde observed:

> The final deliverable of the requirements phase is most often a document in natural language, that in addition to indicative and optative statements may integrate graphical portions of models [...]. A most welcome tool would be one to assist in the generation of such a document [...]

This is exactly what Circe provides by generating (and validating) graphical models from the natural language statements describing the domain (indicative mood) and the requirements (optative[20] mood). The same goal has been pursued by other researchers. We can identify two distinct strands of research:

1. a number of studies assume that requirements are written in completely unrestricted natural language, and use either statistic analysis or general NLP systems to process them;
2. other studies expect requirements written in a more or less restricted or controlled language, using ad-hoc parsing techniques (or traditional context-free grammars) to analyze the text.

In the first group, many proposals have been made in the database and information systems area in the last twenty years (see [11, 12, 18] for early contributions), with regular international symposia (the International Workshop on Applications of Natural Language to Database series, later renamed to International Conference on Applications of Natural Language to Information Systems) being established on this subject in 1995. Due to their specific scope, many contributions have focused in extracting entity-relationship diagrams or comparable static models from an unrestricted natural language description of the expected operations of the system to build and of its domain, possibly using dialog [10] to fill in the missing details and paraphrase [13] to validate the resulting models with the user.

According to published studies, these approaches have been proven to be reasonably effective. However, their wide applicability (stemming from the use of unrestricted NL) is generally obtained at the expenses of the depth and scope of the models that are generated, that are limited to static aspects. In contrast, Circe imposes an additional burden on the specifier (defining a glossary, using controlled NL), but the extra information provided is put to good use in generating and validating richer models on both static and dynamic aspects.

[20]Notice that, as a matter of style, Circe does not use the verbal mood of the statement to distinguish these different roles, but rather the `/SYS` tag. [33] has a good discussion on why this approach is preferable to the one based on moods.

Many of the limitations of those early tools have been overcome by later proposals, focusing mostly on object-oriented modeling. For example, the NLOOPS system [39] uses the general NLP system LOLITA to extract object models from unrestricted NL requirements. Despite the remarkable sophistication and generality of the linguistic analysis stage, these systems are limited by the lack of domain knowledge. Indeed, assuming that the user should provide no further information than the requirements themselves, these systems have to resort to heuristics to identify the proper objects, or rely on domain-specific knowledge bases. In the first case, heuristic algorithms may need to be tuned (e.g., [40], where different sets of classes are extracted from the same text depending on specific numeric parameters of the extraction algorithms), and results may not always be satisfactory. In the second case, the effort to build sufficiently large domain knowledge bases may prove impractical (e.g., [10], where the authors had to interview librarians and library users to build a 12.000 words lexicon on the "library" domain before being able to extract data models from requirements for a library database).

On the contrary, Circe asks the user to provide a glossary for the requirements at hand, which explicitly (through user-visible tags) maps the application domain knowledge provided by the user to the software-systems knowledge provided by the modelers. In our opinion, the extra effort needed to define the glossary is returned manifold under the form of clearer problem descriptions and more reliable synthesized models.

An alternative approach to the identification of significant domain terms, based on statistic properties of a text, is taken by the AbstFinder system [31]. Given a NL requirements document, AbstFinder identifies the repeated text fragments, that with high probability will correspond to significant domain abstractions. Similar but more advanced techniques have been used in the REVERE project [43, 44, 45] to assist in the recovery of models from legacy requirements documents.

Statistic-based techniques cannot guarantee absolute correctness of the results; nonetheless, they provide extremely helpful additions to the human requirements engineer's toolbag, in keeping with the position expressed in [5] that humans should never be put out of the design loop. Circe includes a few modelers that use similar techniques, including a variant of the AbstFinder algorithm (that suggests additions to the glossary taken from often-repeating rests) and a clustering algorithm based on text retrieval techniques used to build a model of the document structure and to suggest ways to improve it.

The second major group of proposals assume that requirements are expressed in a restricted or controlled natural language. Circe belongs to this second category, with MAS rules defining the controlled language accepted. However, the language can be extended by using definitions, to accommodate domain- or problem-specific jargon, as discussed in Section 4.1.

Some author have proposed a fixed language, strictly tied to a particular domain (e.g., AECMA Simplified English [19] for the aerospace domain) or to specific models (e.g., to linear temporal logic in [23]). Very restricted versions of

NL are often comparable to formal languages with NL-like keywords: in [23], the subset of NL used is so strictly defined that there is a 1-to-1 mapping between linguistic structures and linear temporal logic structures. Similar approaches have been taken by a number of researchers, (see, among others, [25, 36, 37, 48]. In some cases (e.g., [37]), a restricted graphical user interface has been used as a mean to enforce strict adherence to the allowable syntax.

Requirements written in such languages can be efficiently transformed into models for the targeted formalism, but are usually very poor sources of information for building models of a different nature. In contrast, the language employed by CIRCE is much more liberal (due to the inherent robustness of the CICO parser and to the capability of extending the language through user-level definitions), and the modularity of the MAS rules and of the expert system makes it possible to extract a much wider array of models from the requirements.

More recently, NLP-based techniques have been proposed as a mean to analyze textual descriptions of use cases rather than full requirements specifications. When used in this context, textual descriptions normally refer to single steps of interaction or computation, while the causal structure of the use case and the set of allowable use cases are described by other means. By reducing the scope of the analysis to single steps, these techniques can attain great precision and good analysis capabilities, as reported in a variety of studies (among them, [7, 22, 47, 46]).

We regard the fact that the research community has shown continuing interest in the automated analysis of natural language requirements as a significant indication of the relevance of this research. CIRCE aims at providing a basic framework and infrastructure that can accommodate most of the ideas put forward in this area. At the same time, CIRCE offers a level of performance, sufficient generality, and an initial set of models and views, that make it usable in industrial practice as well.

9 Conclusions

It has been often observed that natural language is a poor way to represent requirements, due to its inherent leniency towards ambiguity, inconsistency, redundancy, etc. On the other hand, NL is also the language most suited to experimentation and communication — in fact, it is a tool shaped by our minds in millennia of evolution for exactly that purpose.

We believe that natural language is actually well suited to expressing requirements, provided that it is used in a consistent way. For example, a generally unpleasant linguistic phenomenon, ambiguity, can be judiciously used as a very useful abstraction mechanism in expressing requirements — provided that the particular instance of ambiguity *is declared* as abstraction.

The CIRCE framework we presented in this paper proposes such a rigorous use of NL. By structuring requirements documents in three layers (designations, definitions, and requirements), and applying novel parsing and information extraction techniques, precise information can be obtained from NL requirements, while preserving the freedom of expression and ease of communication that is so needed in requirements engineering.

The transformational approach taken by CIRCE combines linguistic techniques, coordination models, and expert system elements in a sophisticated tool that supports the work of the requirements engineer, while maintaining a close interaction cycle that favors rapid experimentation of alternatives and prototyping at the model level. CIRCE can generate models of the requirements document, of the system described by the requirements, and of the requirements development process itself. In this paper, we concentrated on the second group of models, those describing the intended behavior of the system specified in the requirements. A basic set of models, collectively called the Circe Native Meta-Model (CNM), are extracted directly from the requirements, while more sophisticated or derivative models are produced by a set of "software modeling experts" implemented as agents in a modular expert systems. In particular, we have shown how static models (e.g., Entity-Relationship or UML class diagrams) and dynamic models (e.g., finite state automata or event-condition-action rules) are synthesized from the requirements.

The research reported in this paper has opened up a number of avenues for further development. CIRCE provides a comfortable web-based environment for entering requirements and inspecting models and metrics. However, no single tool or environment alone can adequately support the entire software development process. To this end, we plan to provide better integration with other tools for requirements management (e.g., Rational RequisitePro), design (e.g., Rational Rose) and coding (by producing skeleton Java code). As already mentioned, we currently interface with design tools by exporting XMI models. We plan to extend this support by providing an integrated Requirements Development Environment embedded in Eclipse, an open platform for tool integration being developed by the Eclipse Consortium [17].

Scalability issues have not been investigated in depth as yet. CIRCE has been proved on the field to be scalable in terms of concurrent usage (with over 20 teams working on a single server, on different projects, at the same time). However, scalability in terms of requirements documents size has been tested only up to a relatively small 250 requirements. Modular and compositional facilities to handle very large requirements bases must be introduced and tested. Nevertheless, pilot studies conducted by three industrial partners in Italy and Japan have highlighted the considerable benefits provided by CIRCE. These studies pointed out four major improvements over the respective current development practices:

1. the ability of examining, validating, and measuring models of the software to be produced, already at the "informal" requirements stage;

2. the ability to change, in a guaranteedly consistent way, dozens of models with a single change in the text of the requirements;

3. the ability of having immediate feedback over any requirements change, thus making experimentation of alternatives less costly;

4. the ability to start the design phase from a skeleton UML model generated from the requirements and guaranteedly consistent with the same.

Beside the benefits above, pointed out by our industrial partners, we consider two other contributions to be extremely valuable:

5. from our experience in teaching Requirements Engineering and Software Engineering classes, we have found CIRCE to be an effective tool to help develop in the students a special sensibility for the *meaning* of the requirements they write, thus promoting better RE education;

6. the Authors are not immune to learning through experience, either. Many insights and subtle points of various modeling techniques have been uncovered in the process of writing modelers for them. Most of the rules defining the intensional knowledge of the modelers distillate perfectly valid and precise advice that can be followed by humans as well.

So far, CIRCE has been remarkably successful both as a research project and as an implemented tool, both in industrial and in academic environment. In particular, the generality of the framework we designed provides a convenient testbed for developing and testing new modeling and analysis techniques. We hope that this will help promote better requirements engineering practices and advance the state of the art in the field.

References

[1] V. Ambriola and V. Gervasi. Experiences with domain-based parsing of natural language requirements. In G. Fliedl and H. C. Mayr, editors, *Proc. of the 4th International Conference on Applications of Natural Language to Information Systems*, number 129 in OCG Schriftenreihe (Lecture Notes), pages 145–148, June 1999.

[2] V. Ambriola and V. Gervasi. On the parallel refinement of NL requirements and UML diagrams. In *Proc. of the ETAPS 2001 Workshop on Trasformations in UML*, Genova, Italy, April 2001.

[3] V. Ambriola and V. Gervasi. The CIRCE native meta-model. Technical report, Dipartimento di Informatica, Università di Pisa, 2003. (to be published).

[4] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering*, pages 158–165. IEEE CS Press, May 1991.

[5] D. Berry. Natural language and requirements engineering — Nu?, Apr. 2001. Presentation given at the International Workshop on Requirements Engineering, London, UK.

[6] D. Berry and E. Kamsties. The dangerous 'All' in specifications. In *Proceedings of the 10th International Workshop on Software Specification & Design (IWSSD-10)*. IEEE CS Press, 2000.

[7] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A.Maccari. Use case description of requirements for product lines. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL02)*, Essen, Germany, Sept. 2002.

[8] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1982.

[9] E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[10] E. Buchholz, H. Cyriaks, A. Düsterhöft, H. Mehlan, and B. Thalheim. Applying a Natural Language Dialogue Tool for Designing Databases. In *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB'95)*, pages 119–133, Versailles, France, 1995.

[11] P. P. Chen. English sentence structure and entity relationship diagrams. *Information Science*, 29(2):127–149, 1983.

[12] M. Colombetti, G. Guida, and M. Somalvico. NLDA: A natural language reasoning system for the analysis of data base requirements. In S. Ceri, editor, *Methodology and Tools for Data Base Design*, North Holland, 1983.

[13] 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.

[14] D. Damian and D. Zowghi. The impact of stakeholders' geographical distribution on managing requirements in a multi-site organization. *International Requirements Engineering Journal*, 2003. (to be published).

[15] A. M. Davis. *201 Principles of Software Development*. McGraw Hill, 1995.

[16] E. Dubois and K. Pohl. RE 02: A major step toward a mature requirements engineering community. *IEEE Software*, 20(2), January/February 2002.

[17] Eclipse Consortium. Eclipse.org main page. http://www.eclipse.org.

[18] C. Eich. From natural language requirements to good data base definitions — a data base design methodology. In *Proc. of the International Conference on Data Engineering*, pages 324–331, Los Angeles, USA, Apr. 1984.

[19] European Association of Aerospace Industries (AECMA). *AECMA Simplified English (SE) Guide*. available through various AECMA-appointed publishers, Jan. 2001. (The current version is Issue 1 Revision 2; SE has been regularly updated since 1986).

[20] F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi, and S. Ruggieri. Achieving quality in natural language requirements. In *Proc. of the 11th International Software Quality Week*, May 1998.

[21] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. The linguistic approach to the natural language requirements quality: Benefits of the use of an automatic tool. In *26th Annual IEEE Computer Society - NASA Goddard Space Flight Center Software Engineering Workshop*. IEEE CS Press, Nov. 2001.

[22] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Linguistic techniques for use cases analysis. In *Proceedings of the IEEE Joint Internatonal Requirements Engineering Conference (RE02)*, Essen, Germany, Sept. 2002.

[23] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3), 1994.

[24] 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.

[25] N. Fuchs and R. Schwitter. Attempto controlled english (ACE). In *Proceedings of the first international workshop on controlled language applications*, Leuven, Belgium, 1996.

[26] D. Gabbay and A. Hunter. Making inconsistency respectable: A logical framework for inconsistency in reasoning, Part 1 - a position paper. In *Proceedings of Fundamentals AI Research*, pages 19–32. Springer-Verlag, LNCS Volume 535, New York, 1991.

[27] V. Gervasi. *Environment Support for Requirements Writing and Analysis*. PhD thesis, University of Pisa, March 2000.

[28] V. Gervasi. The CICO domain-based parser. Technical Report TR-01-25, Dipartimento di Informatica, Università di Pisa, Nov. 2001.

[29] V. Gervasi. Synthesizing ASMs from natural language requirements. In *Proc. of the 8th EUROCAST Workshop on Abstract State Machines*, pages 212–215, Las Palmas, Spain, February 2001.

[30] V. Gervasi and V. Ambriola. Quantitative assessment of textual complexity. In L. Merlini-Barbaresi, editor, *Complexity in Language and Text*. Edizioni PLUS, Pisa, 2003.

[31] 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.

[32] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the 10th Annual Conference on Computer Assurance (Compass'95)*, pages 109–122, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[33] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. Addison Wesley, 1995.

[34] M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2002.

[35] J. Johnson, K. D. Boucher, K. Connors, and J. Robinson. Project management: The criteria for success. *Software Magazine*, February/March 2001.

[36] B. Macias and S. G. Pulman. *Natural Language Processing for Requirements Specification*, pages 57–89. Chapman and Hall, 1993.

[37] 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.

[38] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

[39] L. Mich. NL-OOPS: From natural language requirements to object oriented requirements using the natural language processing system LOLITA. *Journal of Natural Language Engineering*, 2(2):161–187, 1996.

[40] L. Mich, J. Mylopoulos, and N. Zeni. Improving the quality of conceptual models with NLP tools: An experiment. Technical Report DIT-02-0047, Department of Information and Communication Technology, University of Trento, 2002.

[41] B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the* $22^n d$ *International Conference on Software Engineering (ICSE 2000)*, Limerick, June 2000. ACM Press.

[42] Object Management Group. *OMG-XML Metadata Interchange (XMI) Specification, v1.2.* 2002. http://www.omg.org/technology/documents/formal/xmi.htm.

[43] P. S. P. Rayson, R. Garside. Recovery legacy requirements. In *Proceedings of the Fifth International Workshop on Requirements Engineering: Foundations for Software Quality*, Heidelberg, Germany, 1999.

[44] P. Rayson, R. Garside, and P. Sawyer. Language engineering for the recovery of requirements from legacy documents. Technical Report REVERE Project, Lancaster University, May 1999.

[45] P. Rayson, R. Garside, and P. Sawyer. Assisting requirements engineering with semantic document analysis. In *Proceedings of the Computer-Assisted Information Retrieval Conference (RIAO 2000)*, pages 1363–1371, Paris, Apr. 2000.

[46] D. Richards and K. Boettger. Representing requirements in natural language as concept lattices. In *Proceedings of the 22nd Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (SGES)*, Cambridge, UK, Dec. 2002.

[47] D. Richards, K. Boettger, and O. Aguilera. A controlled language to assist conversion of use case descriptions. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence*, Canberra, Australia, Dec. 2002.

[48] C. Rolland and C. Proix. A natural language approach for requirements engineering. In P. Loucopoulos, editor, *Advanced Information Systems Engineering*, number 593 in LNCS. Springer-Verlag, 1992.

[49] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[50] H. Schmid. Probabilistic part–of–speech tagging using decision trees. In *Proceedings of the Conference on New Methods in Language Processing*, pages 44–49, Manchester, UK, Sept. 1994.

[51] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the* $22^n d$ *International Conference on Software Engineering (ICSE 2000)*, Limerick, June 2000. ACM Press.

[52] D. Zowghi. Does global software development need a different requirements engineering process? In *Proceeding of International Workshop on Global Software Development*, Orlando, Florida, 2002. (In conjunction with the Int'l Conference on Software Engineering, ICSE 2002).

Contents

List of Figures

List of Tables

List of Algorithms