Automated Software Engineering, 13, 107–167, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

On the Systematic Analysis of Natural Language Requirements with CIRCE

VINCENZO AMBRIOLA ambriola@di.unipi.it
VINCENZO GERVASI* gervasi@di.unipi.it
Dipartimento di Informatica—Università di Pisa, Italy

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 then 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. The result of all the transformations is a set of models for the requirements document, for the system described by the requirements, and for the requirements writing process. These models can be inspected, measured, and validated against a given set of criteria.

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 20 years. Since it was first identified as an autonomous discipline, its role and importance in the software development process have never been refuted. From the first estimates (Boehm, 1982) to recent studies (Johnson et al., 2001), 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 (Dubois and Pohl, 2002), has led to the structuring of the discipline into a number of well-defined sub-areas. Nuseibeh and Easterbrook (2000) name *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* (van Lamsweerde, 2000); a number of other cognitive, social and organizational issues are also sometime referred to as part of RE. 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 (Choi et al., 2002; Eshuis et al., 2002) and SCR* (Heitmeyer et al., 1995)) are nowadays reasonably common among practitioners.

*Corresponding author. phone: +39 050 2212773; fax +39 050 2212726.

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. In fact, in the course of our research, we have extended the environment several times, and customized it in order to support formal specification techniques that were already in use by some of our industrial partners (e.g., Gervasi, 2001b, 2002).

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, 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. This leaves a requirements engineer free to choose the process that is most suited to a particular task, and still be able to use his toolbox, rather than forcing the same tool-defined process on all problems. However, we provide in Section 7 an example case showing how CIRCE can be applied in a variety of different situations. The work presented here is the result of several years of research activity, and reports on other facets of this research have already been published, starting with Ambriola and Gervasi (1997). In particular, the interested reader can find more details on process measurement and management issues with CIRCE in Ambriola and Gervasi (2000), and a full report on an industrial case study in Gervasi and Nuseibeh (2000, 2002). A summary of early results was provided in Gervasi (2000). Other relevant references are noted in due course in the following.

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 detail in Sections 4 and 5, concerning respectively the parsing of NL requirements and 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. The example 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 analyst could immediately start asking relevant questions. 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. The analyst could extract and present a data flow model from the requirements, like the one shown in Figure 1(a). Looking at that graphical representation, the customer or the analyst could realize that navigation commands should also take height and heading into consideration, and possibly other information

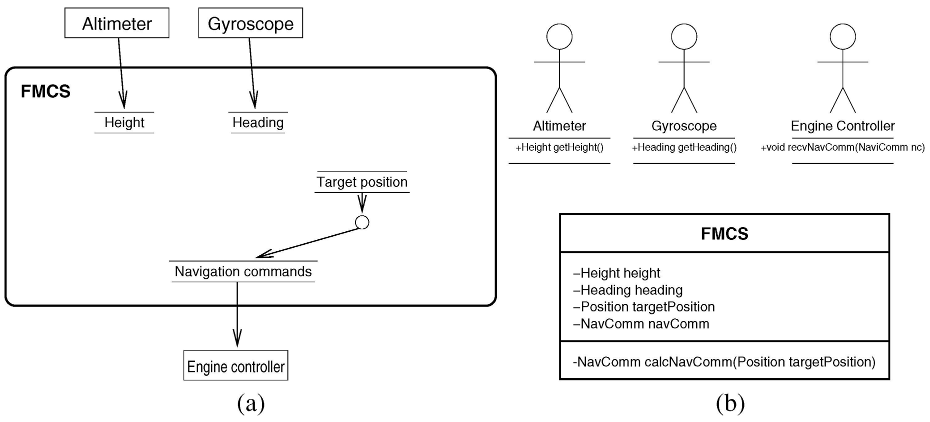


Figure 1. (a) A data flow model and (b) a UML object diagram extracted from the sample requirements for the FMCS.

as well. Requirements could be added to correct these and other problems, and more requirements to describe other functions of our FMCS could be written.

As our requirements document grows, more and more information becomes available. Through a careful scrutiny of text, we could be able to extract information from the requirements and reorganize it into other forms, more suitable for specific tasks. We could for example produce UML models that are synthesized from the requirements (like, for example, those in Figure 1(b)).

However, performing all these analyses is time-consuming and error-prone, not to mention being terribly boring, especially if repeated over and over again every time new requirements are added or old requirements are changed. The time and ability of highly specialized professional requirements engineers should not be wasted in repeating work that could be automatized. The requirements analysis process could be made more efficient if the *systematic* parts of the process are automatized, leaving the engineer free to exercise his best qualities—inventiveness, social sensibility, inquisitiveness, aestethic sense of design—on more interesting activities. In this paper we advocate that systematic activities in requirements modeling and analysis should be supported by automatic tools, which can provide facilities such as building graphical models, validating them and measuring some of their features.

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 2: the spikes in the functional

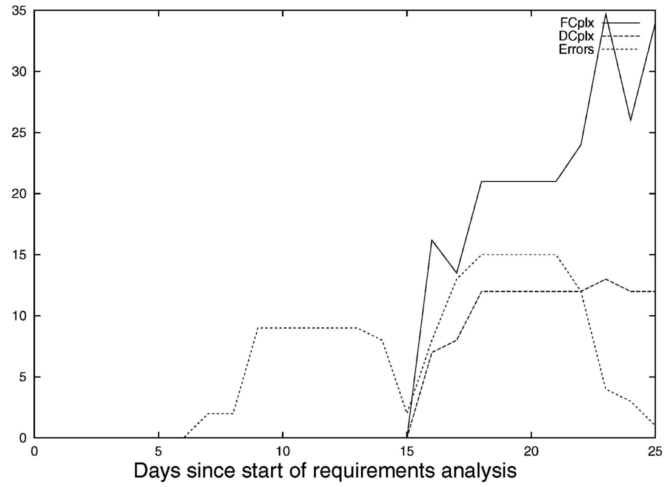


Figure 2. Process diagrams for a requirements development process. Plotted in the diagram are measures of functional complexity (FCplx), data complexity (DCplx) and the number of errors found in the requirements at any given time (Errors).

and behavioral complexity scores warn us that the requirements are changing a lot, and are still *semantically* in a state of flux—i.e., editing on the requirements is not just cosmetic, but really impacts the functionality described therein. Moreover, the relatively high number of unresolved errors and warnings as the deadline approaches, on day 25, constitutes a serious indication that the process may not be managed well.

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, 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(a) and (b) from our previous example, and
- models of the *requirements development process*, through the traces of relevant artifacts, as in Figure 2 from our example.

From a user's perspective, CIRCE is a work companion that reads natural language requirements as input, and—upon the 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, possibly according to the suggestions provided by CIRCE. 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.

The general architecture of CIRCE is shown in Figure 3. 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 described fully in Section 4.

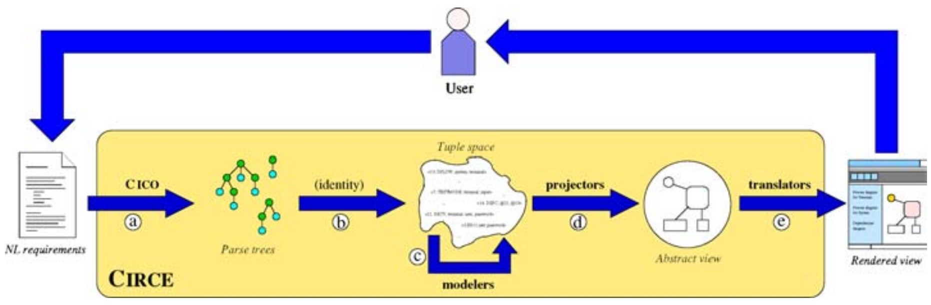


Figure 3. A general overview of the transformations on NL requirements operated by CIRCE.

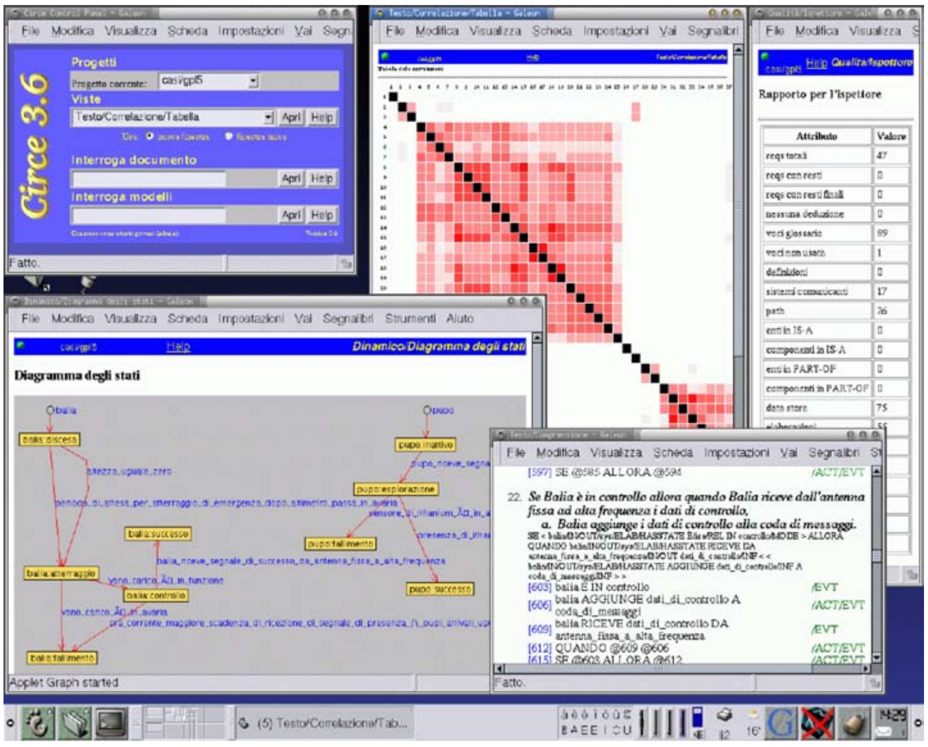
ⓑ The parse trees obtained from the previous transformation are then encoded as tuples and stored in a shared tuple space. In CIRCE, this encoding is performed by CICO alongside with the parsing process: as a consequence, this step can be considered formally to consist in an identity transformation. 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 called *modelers*. CIRCE includes a library of over one hundred modelers, covering such diverse aspects as static, functional and behavioral modeling, validation and measurement 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, 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.

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, a user can edit his requirements and ask for specific views. For each request, CIRCE performs the transformations ⓐ–ⓔ needed to generate the requested view, and sends back the result as a



Rapporto per l'ispettore

Attributo	Valore
reqs totali	47
reqs con resti	0
reqs con resti finali	0
nessuna deduzione	0
voci glossario	89
voci non usate	1
definizioni	0
sistemi comunicanti	17
path	26
enti in IS-A	0
componenti in IS-A	0
enti in PART-OF	0
componenti in PART-OF	0
data store	75
elaborazioni	55

[597] SE @585 ALLORA @594 /ACT/EVT

22. *Se Balia è in controllo allora quando Balia riceve dall'antenna fissa ad alta frequenza i dati di controllo,*
   a. *Balia aggiunge i dati di controllo alla coda di messaggi.*

SE < balia/INOUT/sys/ELAB/HASSTATE Bäs/REL IN controllo/MODE > ALLORA QUANDO balia/INOUT/sys/ELAB/HASSTATE RICEVE DA antenna_fissa_a_alta_frequenza/INOUT dati_di_controllo/INF < < balia/INOUT/sys/ELAB/HASSTATE AGGIUNGE dati_di_controllo/INF A coda_di_messaggi/INF > >

[603] balia E IN controllo /EVT
[606] balia AGGIUNGE dati_di_controllo A coda_di_messaggi /ACT/EVT
[609] balia RICEVE dati_di_controllo DA antenna_fissa_a_alta_frequenza /EVT
[612] QUANDO @609 @606 /ACT/EVT
[615] SE @603 ALLORA @612 /ACT/EVT

Figure 4. A sample session with CIRCE. Going clockwise from the top left corner, the following windows are open: (1) the Portal, through which a user can ask for specific views on the requirements to be opened; (2) the Text Correlation view, graphically showing the degree of textual correlation between all pairs of requirements; (3) the Inspector view, showing values for a number of metrics collected on the requirements; (4) the Parsing Report, providing a paraphrase of how requirements were parsed by CIRCE; (5) the Finite State Automata view, describing graphically the various states in which systems described in the requirements can be, and associated transitions and conditions. The figure also shows the multilingual support in CIRCE: Italian is used in this case. See Section 7 for more screenshots in English.

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 4. 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, that is a requirements document, and output artifact, that is a forest of parse trees. We then present in some detail the specific parsing algorithms that are used in CIRCE; the interested reader can refer to Gervasi (2001a) for a fuller treatment.

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 M. Jackson (2002), designations in $\mathcal{G}$ establish specific *names* for entities or phenomena that are relevant in the domain being considered. Each term in $\mathcal{G}$ is *tagged* with labels declaring its properties. In addition, a designation explicitly establishes an equivalence between different terms that refer to the same entity, e.g., synonyms or metonyms.[1]

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 it 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 as terminal symbols 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 a thing 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 $\mapsto$symbol, which is read as "is defined as", 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 but 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 refers only to primitive terms and concepts. These canonical 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, 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 or not the subsystem x that we are considering is part of the system we are going to design.

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 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.

4.2.1. Typography and formatting. Typographical and formatting details are handled by reducing the source text into a *canonical form* by:

- normalizing typographic variants of certain symbols;
- substituting special characters (e.g., ß) and punctuation (e.g., $\ll$ and $\gg$) with corresponding canonic symbols;
- inserting markup symbols to represent structural features of the text (e.g., section structure or itemization);
- discarding comment lines and other whitespace.

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:

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

4.2.2. 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[3]

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

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

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

Glossary substitution is obtained by applying the $map_{\mathcal{G}}()$ function defined below to each ω_i in each requirement in $\mathcal{R}$:

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

A number of *well-formedness rules* apply to glossaries (for example, to guarantee that a term is not declared as synonym for two different terms); due to space considerations, we omit them here (the interested reader can refer to Gervasi (2001a) for full details).

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

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

4.2.3. Morphosyntactic analysis. Morphosyntactic analysis on the text is performed by using TreeTagger (Schmid, 1994), a part-of-speech (POS) 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 (Marcus et al., 1993).

For simplicity, we will not use POS tags in our examples. However, in specific cases POS tags can be important to distinguish between multiple meanings or roles of a word. Except for the way they are assigned to words, POS tags are treated in CIRCE exactly as glossary-based tags.

4.3. Parsing

Given a text tagged and canonized as shown above, CIRCE parses the text by using the CICO domain based parser (Ambriola and Gervasi, 1999; Gervasi, 2000, 2001a).

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

$\mu = \langle \tau_1, \ldots, \tau_n \rangle$ is the *model*,
$\alpha = \langle \beta_1, \ldots, \beta_p \rangle$ is the *action*, and
$\sigma = \langle \gamma_1, \ldots, \gamma_q \rangle$ is the *substitution*.

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

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

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, often hinting at non-functional requirements, 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

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

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 $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, 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[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 $\sigma' = \langle \gamma'_1, \ldots, \gamma'_q \rangle$ is defined by

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

where

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

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 $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, 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

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

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

4.3.6. 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 after other parts have been taken away 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. A number of structural features in the source text can introduce distinct *parsing contexts* in a statement. Itemized lists, parentheses or headings and sub-headings all introduce a local scope for parsing purposes. Deeply nested contexts of this kind are particularly common in requirements documents, whose sectional structure often mirrors the nesting of concerns according to a top-down description style. We will use $\triangleleft$ and $\triangleright$ markers to indicate a parsing context, like in

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

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

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}
$f$ := false
**for all** $\rho \in R$ **do**
  **if** match$(\rho, s)$ is complete **then**
    **for all** $m$ valid matches from match$(\rho, s)$ **do**
      $f$ := true
      $(s', A') :=$ apply$(\rho, m, (s, A))$
      $k' := k +$ mscore$(m)$ + 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}
  $k' := k +$ lscore$(s)$
  **if** $k' > B_k$ **then**
    $B_k := k'; B_A := A; B_s := s$
  **end if**
**end if**
{$B_k$, $B_A$, and $B_s$ are respectively the score, the action sequence and the final rest of the best parse
sequence, according to the scoring system}
```

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

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

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

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

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

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

Algorithm 3 ContextParse(s, R).

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

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:

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

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

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,

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
```

syntactic structures that cannot be parsed in the local scope are recursively resolved against the enclosing scope, until a valid match is found.

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:

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

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
```

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.

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 or 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 the matched term 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.

Space considerations prevent us from describing all these forms of tagging in detail here. The interested reader can refer to Gervasi (2001a) for a complete formal treatment.

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[6]

```
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:

$$\mathsf{match}(\rho_1, s) = \langle \underbrace{\{system, base\}}_{snd}, \underbrace{\{send\}}_{SEND}, \underbrace{\{heartbeat\}}_{data}, \underbrace{\{to\}}_{TO}, \underbrace{\{system, base\}}_{rcv} \rangle$$

$$\mathsf{match}(\rho_2, s) = \langle \underbrace{\{every\}}_{EVERY}, \underbrace{\emptyset}_{period}, \underbrace{\emptyset}_{action} \rangle$$

$$\mathsf{match}(\rho_3, s) = \langle \underbrace{\{10\}}_{number}, \underbrace{\{second\}}_{SECOND} \rangle$$

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:

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

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 $sa_1 = \text{apply}(\rho_1, m_1^a, (s, \langle\rangle))$, obtaining the new sa-structure

```
sa₁ = ( ⟨the/DT @₁/ACT/EVT every/DT 10/JJ/CD second/NN/S⟩,
        ⟨⟨@₁,SEND system/... base/... heartbeat/...⟩⟩
      )
```

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 step thus produces

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

Rule ρ_2 is now applicable, with match

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

Thus producing

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

No further rules are applicable, and the parse tree thus obtained (that is shown in Figure 5) 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

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

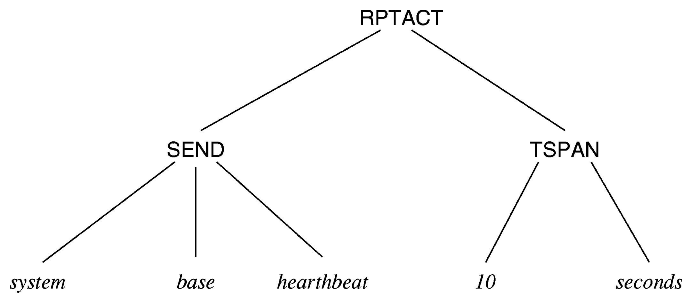


Figure 5. The parse tree for the requirement "The system shall send a heartbeat to the base every 10 seconds".

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 (Fabbrini et al., 1998). For example, grammatical errors may be revealed by parsing failures, and the presence of potentially ambiguous words like "some" or "reasonably" (see for example (Berry and Kamsties, 2000; Fabbrini et al., 2001)) can be checked at parsing time by simply writing a few MAS rules to catch the offending keywords.

5. Model building & visualization

The previous section has discussed the transformation marked with ⓐ in Figure 3, 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 and of displaying them to the user in a convenient fashion is described by the transformations marked ⓒ, ⓓ and ⓔ in Figure 3, and 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. Notice that the specification is given *operationally*: while in principle having a declarative (e.g., rule-based) specification seems nice, in our experience pure rule-based mechanisms (like those found in some early expert systems) have proven to be too limiting for our purposes. For example, a few modelers perform numerical matrix computations, that are not easily implemented in rule-based languages. Also, the unit of computation for modelers is typically an entire T-model, not a single tuple: in this sense, modelers can reason on entire models rather than being limited to simple if-then rules.

A modeler must satisfy the three correctness properties below:

(m1) In order to ensure non-retractability[7] of requirements and of models, 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,

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

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

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

where SEND and RECEIVE[8] 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:

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

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:

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

We disallow cyclic dependencies between modelers[9], 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 $m_1 < m_2 < \cdots < m_n$ 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) measurement, (iv) validation, and (v) visualization support.

The simplest among the modelers are the *abstraction modelers*. Modelers in this class 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

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

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, an abstraction modeler performs also 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*. An intensional modeler combines 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 presented in Section 7.7 that build a UML model of the system described by the NL requirements belong to this class, as do most other modelers.

The third class of modelers are the *measurement modelers*. Their purpose is to *measure* attributes of other models, as expressed by the tuples in the tuple space. This measurement may include counting occurrences of certain tuples, as well as more complex measures. For example, a modeler in this class collects function point counts for each subsystem, based on their I/O specification and on the internal computation described in the requirements, and provides estimates for related attributes. As for all the other modelers, the measures collected by measurement modelers constitute a T-model, and are thus put back in the tuple space, where they can be analyzed by other modelers. A number of measurement modelers build models of the requirements development process based on the historical evolution of the requirements document. These models and related measures are used to identify potentially dangerous development situations, in addition to being shown 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*. Each modeler in this class checks that certain correctness properties on some T-model are satisfied, and emits new tuples with the results of such checks. The properties that are checked for may include simple intra-T-model checks or more complex inter-T-model checks. For example, an intra-T-model check concerns so-called *dead data*: in the data flow model, any data store whose value is never used is suspicious, and finding such a data store may hint at a missing requirement. An example of inter-T-model check is the *invalid selection* check: 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 the considered 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 continuously 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*. This class includes modelers responsible for manipulating 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. Each view support modeler generates 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 a single kind of software component (the modeler) and a uniform data structure (the T-model) for such broad range of tasks offers many advantages.

In terms of architecture, such minimality makes CIRCE simple to understand and maintain: the kernel of the system has only to schedule the computation of the various modelers in response to the queries that come from a user. Moreover, it makes CIRCE extremely expandable: by writing a new modeler, that often amounts to a handful of lines of 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 collectively 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. A measurement modeler examines the output of the validation modelers in order to obtain a measure of the number of potential errors in the requirements. An intensional modeler builds on top of these measures to obtain a T-model of the requirements development process. Other measurement modelers look at the process T-model, and provide estimates of such attributes as stability, maturity, etc. A higher-level validation modeler looks 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.

The free combination of abstraction, modeling, measurement, validation, and presentation is used throughout in CIRCE in a variety of contexts. It is not unreasonable to think of the mixing of all these activities in CIRCE as the mirroring—albeit partial and approximate—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 Goldin and Berry (1997) put it, every attempt to put the human out of the loop in RE is fundamentally flawed.

In CIRCE, several 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.[10]

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*,[11] and (ii) *translation* of the commands into *rendering instructions*. These two steps correspond to the transformations ⓓ and ⓔ in Figure 3, 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 $f : \mathcal{T} \rightarrow \mathcal{L}_t$ 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:

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

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

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
```

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 $f : \mathcal{L} \to \mathcal{H}$ is the interpretation function that maps a description in the representation language t into the final rendered page (we use $\mathcal{H}$ here to indicate the domain of valid HTML pages).

CIRCE includes a number of translators for most common representation styles. These include editable or read-only text, forms, tables, graphs and histograms; a translator for downloadable files is also provided to facilitate the export of generated models (e.g., XMI files for UML models).

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} can be described formally as

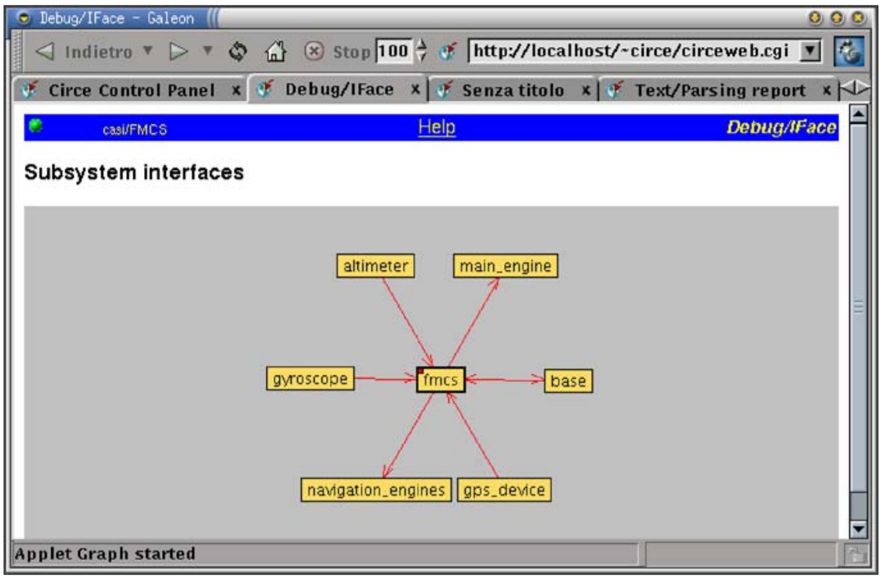
$$p_{iface} = \langle\{IIFACE, OIFACE\}, \text{“Debug/IFace”}, f, \mathsf{graph}\rangle$$

where f is the function expressed by Algorithm 6.

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 6.

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 view, that can be observed in the upper-left corner in Figure 4 (page 113), 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 view is provided 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.



Subsystem interfaces

Figure 6. The final rendering of the view generated by v_{iface}.

When the user selects a view v to display[12] 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 software needs to be installed on the client machine. This is particularly important in global software development (Zowghi, 2002; Damian and Zowghi, 2003), especially during requirements negotiation;
- the user interface 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 programmatically generate a static web site for a project, containing for example (i) all the source documentation related to the requirements, including requirements, definitions, designations; (ii) textual, tabular, or graphical representations of all the models that are deemed relevant; (iii) a full report of the validation checks performed and of their outcome.

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

The previous two sections 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, and we have not discussed which specific models are synthesized by our modelers and especially by the intensional modelers discussed in 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.[13]

Due to space considerations, we do not provide a full formal definition of the CNM in this work,[14] and in particular do not discuss document or process models, nor elaborate on measurement and validation of the CNM. However, the following describes 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.

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 each of these aspects in turn.

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 application 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[15]—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 are. For example, the attributes *weight* and *fuel load* of a missile are not independent: if the fuel tank of a missile is loaded with fuel, its weight will be increased by the weight of the fuel.

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—called a *reification* or hypostatisation—of the observed entity. It is the responsibility of the specifier to describe how this internal representation is kept synchronized 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-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.

A relationship is usually 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 relationship, possibly by having it 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 that it is a dynamic relationship. In this case, we can explicitly introduce the names-coordinates table and possibly describe in the requirements how and when it is updated by the command center operator.

6.1.2. Instance selection. To identify particular instances from a class of similar entities, a *selective expression* can be used. A selective expression includes a linguistic *quantifier* (e.g., "a", "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

Table 1. Some selective expression and their meaning in CNM.

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

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 $\forall x, \lvert target_R(x) \rvert = 1$. To encourage precise writing, a selective expression using a determinative quantifier (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 a path consisting entirely of 1-navigable relationships. Other uses are also accepted, but flagged with a warning suggesting the use of indeterminative quantifiers instead.

As an example, Table 1 lists some selective expressions, together with an explanation of how they are interpreted in CNM. Other examples are provided in Section 7.

6.2. *Agents and dynamic aspects*

An agent in CNM can be active, reactive or passive, and can communicate with other agents either explicitly by exchanging messages, or implicitly by observing some shared portion of their state. Each active agent has its own flow of control, that often will correspond to a separate thread or process in the implemented system; it can take autonomous actions and initiate chains of causal phenomena. A reactive agent is similar, in that it can react to events that it observes in the world, but can never initiate a causal chain on its own—it can only take part in such a chain as an intermediate of final link. In the implemented system, reactive agents often corresponds to code which is executed on someone else's thread, or to a mostly sleeping process. A passive agent is an even more restricted form of agent: it can change its state in response to some external event, but can never propagate a causal chain by taking further actions—hence, it can only appear as a final link in a causal chain. Passive agents are often used to serve as a storage area for the implemented system, e.g., a DBMS module.

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

Events, conditions, and actions can be given 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 2 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:

> 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

$$\begin{aligned} r = \langle\ & \text{FMCS enters operation mode}^e, \\ & \text{FMCS turns on the main engine}^a, \\ & \text{engine is ignited correctly}^c, \end{aligned}$$

Table 2. An example of how linguistic context influences the event, condition, or action role of an expression.

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

ground speed exceeds 20 Km/h^e,
FMCS enters flight mode^a⟩

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. An ECA rule that has an event as its first element is called *normal*; the event is said to be the *head event*. Non-normal rules can be *normalized* according to the rules below:

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 or normalized ECA rules. At each computation step, the agent verifies which ones among the head events of its rules have 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 (Börger and Stärk, 2003), of which our model is a superset (Gervasi, 2001b).

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 3.

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

Table 3. Some of the special linguistics forms to express various kinds of computations.

Expression	Definition
appends x to y	$y' = y \oplus x$
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	$y' = y \ominus x$
extracts x from y	$y' = y \ominus \mathit{first}(y) \wedge x' = \mathit{first}(y)$
updates y using x	$y' = f(y, x)$

Legend

x'	new value of x after the computation
f	unspecified function
$y \oplus x$	the collection y, with the element x added
$y \ominus x$	the collection y, with the element x removed
$\mathit{first}(y)$	a distinguished element of the collection y

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.[16] In addition to that, and to abstract events declared with /EVT in $\mathcal{G}$, a number of other events can be expressed.

Monadic, dyadic, 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.

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. /TSPANs 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 4 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 of 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.

Table 4. Time-related expressions in CNM.

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[17] 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

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—not to mention space concerns—in this example we do not investigate what happens after that stage, but we can imagine that the control system would enter a different state, controlling the cruise until an assigned target is reached. 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 certain domain elements exist, and agree on their 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 such as is-a and part-of.

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 certain, 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, or the parser will complain, 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, making decisions, and autonomously changing its state. Each of these capabilities requires 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 7 shows a state transition diagram representing the various states in which the FMCS can be, together with the respective transitions. As can be observed in the figure, 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 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

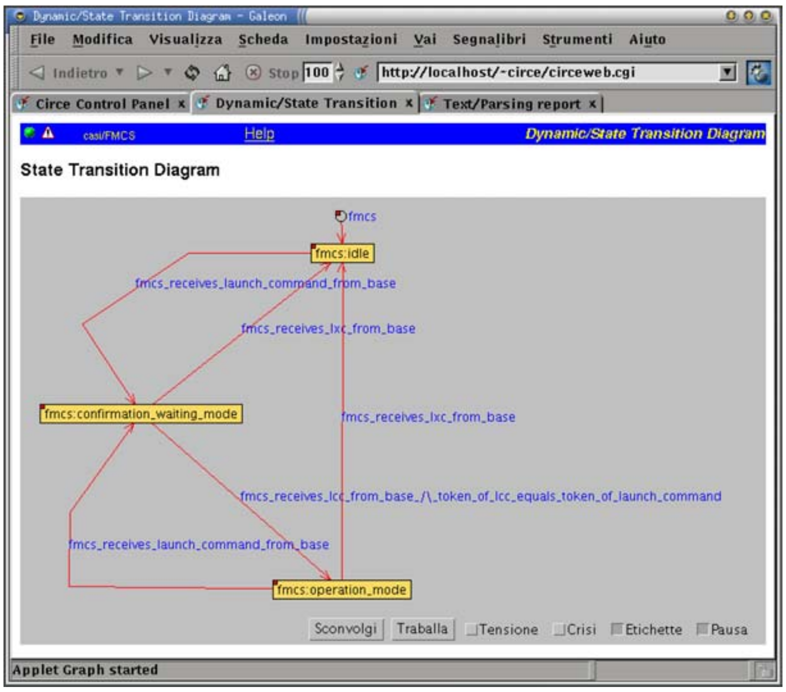

Dynamic/State Transition Diagram - Galeon
File Modifica Visualizza Scheda Impostazioni Vai Segnalibri Strumenti Aiuto
Indietro Stop 100 http://localhost/~circe/circeweb.cgi
Circe Control Panel
Dynamic/State Transition
Text/Parsing report
casi/FMCS
Help
Dynamic/State Transition Diagram
State Transition Diagram
fmcs
fmcs:idle
fmcs_receives_launch_command_from_base
fmcs_receives_lxc_from_base
fmcs:confirmation_waiting_mode
fmcs_receives_lxc_from_base
fmcs_receives_lcc_from_base_/_token_of_lcc_equals_token_of_launch_command
fmcs_receives_launch_command_from_base
fmcs:operation_mode
Sconvolgi
Traballa
Tensione
Crisi
Etichette
Pausa
Applet Graph started

Figure 7. A state transition diagram synthesized from our initial requirements.

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 something 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   ↦  SEND AN ACTIVATION COMMAND TO x
TURN x OFF  ↦  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, e.g., `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 commands 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 purposely 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 altitude of 30 meters. These declarations must be made by writing:

Definitions

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

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,[18] and has additional advantages in facilitating the evolution of the requirements. For example, let us suppose that at some future time the problem owner decides 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 already written, by changing 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, 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* of 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 8(a) shows a view depicting the data flows between the various subsystems in our domain. We can see and possibly 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 setting of the communication channels is actually what we would have expected; hence, we can rest assured that this particular aspect of our requirements is correct.

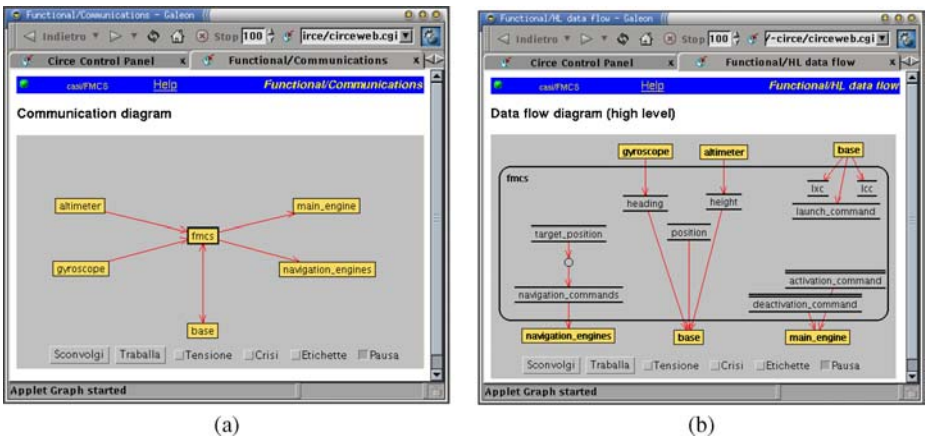


Figure 8. (a) The Communications model for the sample requirements. (b) The High-Level Data Flow model for the same requirements.

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 in the Pandora view the following warning, among others:

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.
Relevant views: Functional/HL data flow, Functional/LL data flow.

By looking at the high-level data flow diagram in Figure 8(b), as suggested by the warning message, 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, as declared in Section 7.2. Additional 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, explicitly postponing its resolution to a later stage, as has been often suggested (Balzer, 1991; Gabbay and Hunter, 1991; Finkelstein et al., 1994), with the assurance of a continuous reminder to resolve the problem.

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 the growing Entity-Relationship model, shown in Figure 9, providing 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 10 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

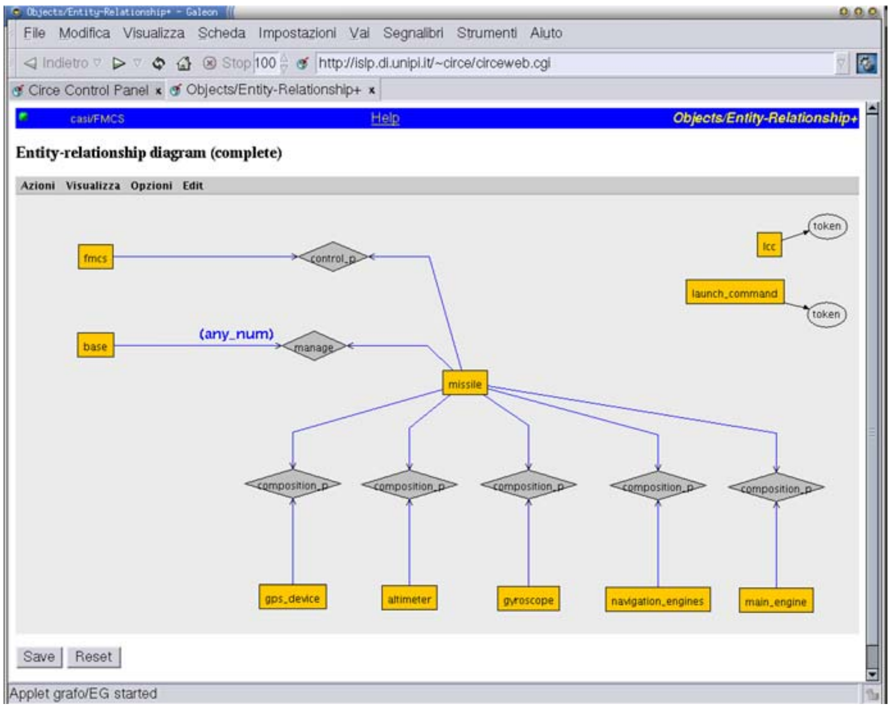

Objects/Entity-Relationship+ - Galeon
File Modifica Visualizza Scheda Impostazioni Vai Segnalibri Strumenti Aiuto
Indietro Stop 100 http://islp.di.unipi.it/~circe/circeweb.cgi
Circe Control Panel
Objects/Entity-Relationship+
casi/FMCS
Help
Objects/Entity-Relationship+
Entity-relationship diagram (complete)
Azioni Visualizza Opzioni Edit
fmcs
control_p
base
(any_num)
manage
missile
lcc
token
launch_command
token
composition_p
composition_p
composition_p
composition_p
composition_p
gps_device
altimeter
gyroscope
navigation_engines
main_engine
Save Reset
Applet grafo/EG started

Figure 9. The Entity-Relationship model for the sample requirements.

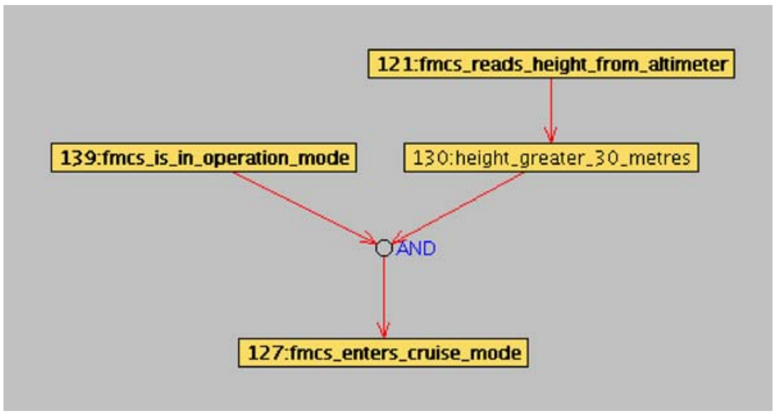

121:fmcs_reads_height_from_altimeter
139:fmcs_is_in_operation_mode
130:height_greater_30_metres
AND
127:fmcs_enters_cruise_mode

Figure 10. 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.

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

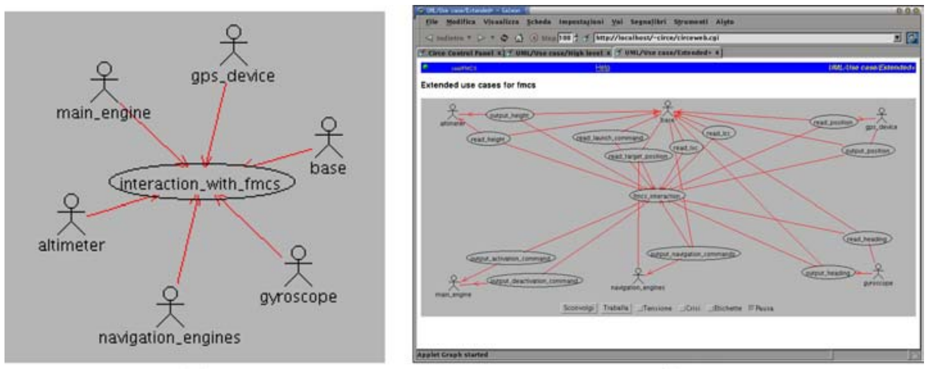
Our requirements for the FMCS up to this point, albeit of course far from complete, are already enough to start thinking about a design and a prototype for the final implementation. CIRCE comes to help in this respect by providing views for UML Use Case Diagrams, Class Diagrams, Sequence Diagrams, Collaboration Diagrams, and State Transitions Diagrams (Rumbaugh et al., 1998; Ambriola and Gervasi, 2001). In addition, CIRCE can generate pseudo-code for the system described by the requirements, in the form of Abstract State Machines (Börger and Stärk, 2003; Gervasi, 2001b) specifying the behavior of each agent.

As an example, Figure 11 shows the high-level and detailed view of the use cases identified by CIRCE in our requirements, whereas Figure 12 shows the UML Class Diagram generated from the requirements.

If we are satisfied with the resulting models, we can directly export them through XMI (Object Management Group, 2002) to other tools, e.g., Rational Rose, and continue the detailed design activity in a more specific environment. Alternatively, 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, indicated 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.



(a) (b)

Figure 11. High-level (a) and detailed (b) use cases from the FMCS requirements.

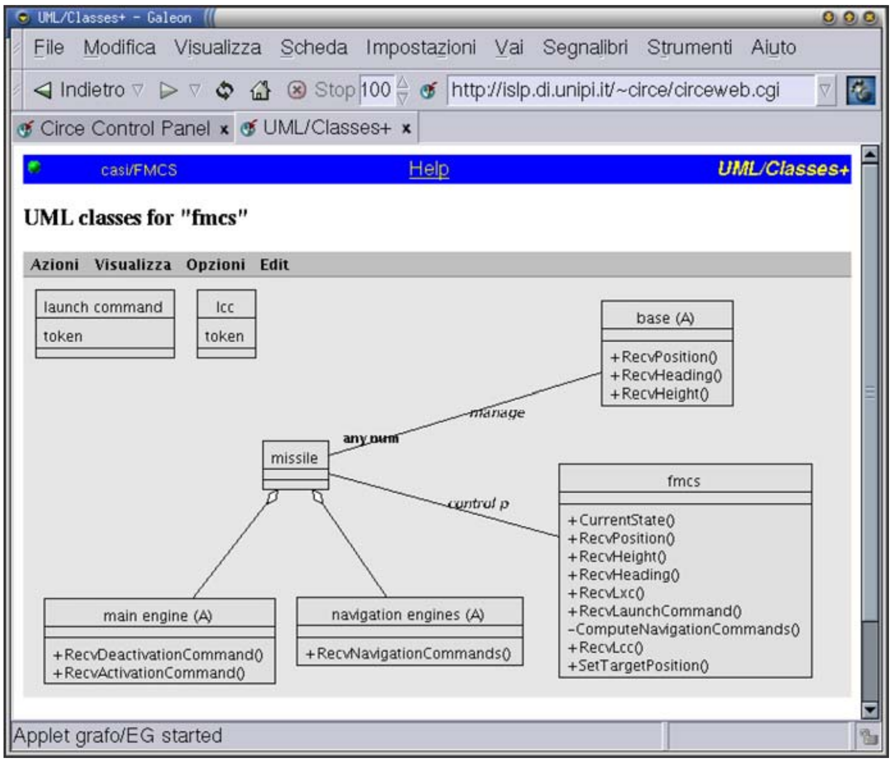


Figure 12. UML Class Diagram synthesized by CIRCE from the FMCS requirements.

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 $5\textsc{SecondsAfterFMCSEntersOperationMode}^e$ and $State(self) = OperationMode$
then
    $\textsc{Send}(MainEngine, deactivation\_command)$
    $State(self) := Idle$
end if
if $20\textsc{Milliseconds}^e$ and $State(self) = CruiseMode$ then
    $\textsc{Recv}(Altimeter, height)$
    $\textsc{Recv}(Gyroscope, heading)$
    $navigationCommands := f(targetPosition)$
    step
    $\textsc{Send}(NavigationEngines, navigationCommands)$
end if
if $2\textsc{Seconds}^e$ and $State(self) = CruiseMode$ then
    $\textsc{Send}(Base, position, height, heading)$
end if
```

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 requirements analysis by encouraging the use of a definite writing style, by reporting certain syntactic and semantic 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.

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. Recent, independent surveys (Mich et al., 2004; Neill and Laplante, 2003) have found that between 93% and 95% of all requirements documents in industrial practice are written in natural language, with structured natural language accounting for some 15% of the total. In Davis (1995), Davis listed among his principles of Software Engineering: "Increase, never substitute, natural language". Five years later, in his 2000 review and agenda paper (van Lamsweerde, 2000), 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 [...]

CIRCE is an attempt to satisfy van Lamsweerde's call. It assists the requirements engineer by validating the natural language statements describing the domain (indicative statements) and the requirements (optative statements[19]) in terms of syntax and, to a certain extent, of semantics. Moreover, further assistance is provided by generating tabular and graphical models from these statements, and by computing related measures. 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 Abbot, 1983; Chen, 1983; Colombetti et al., 1983; Eich, 1984) for early contributions), with regular international symposia such as the International Workshop on Applications of Natural Language to Database, 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 (Buchholz et al., 1995) to fill in the missing details and paraphrase (Dalianis, 1992) 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 by requiring the definition of a glossary and the use of controlled NL, but the extra information provided is put to good use in generating and validating richer models on both static and dynamic aspects.

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 (Mich, 1996) 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., Mich et al., 2002), 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., (Buchholz et al., 1995), where the authors had to interview librarians and library users to build a 12.000 words lexicon about 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, annotated with user-visible tags, which maps on the fly the application domain knowledge provided by the user to the software-systems knowledge provided by the modelers. In our experience, 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 (Goldin and Berry, 1997). 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 (Rayson and Garside, 1999; Rayson et al., 1999, 2000) 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 by Berry (Goldin and Berry, 1997) 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 its MAS rules defining the controlled language accepted. However, the language can be extended through definitions, to accommodate domain- or problem-specific jargon, as shown in Section 4.1.

Some author have proposed a fixed language, strictly tied to a particular domain, e.g., AECMA Simplified English (AECMA, 2001) for the aerospace domain, or to specific models, e.g., to linear temporal logic in Fantechi et al. (1994). Very restricted versions of NL are often comparable to formal languages with NL-like keywords: in Fantechi et al. (1994), 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 (Fuchs and Schwertel, 2003; Schwitter, 2002), where discourse representation structures and first-order logic are used; Sukkarieh and Pulman (1999) where specifications are mapped to a special purpose logic; or the older (Rolland and Proix, 1992). In some cases, a restricted graphical user interface, as in Macias and Pulman (1995), or a syntax-guided editor as in Schwitter et al. (2003) have been used as a mean to enforce strict adherence to the allowable language.

Requirements written in such languages can be efficiently transformed into models for the targeted formalism, but are usually very poor sources of information when it comes to building models of a different nature. For example, the language used by Fantechi et al. in Fantechi et al. (1994) allows the expression of linear temporal logic properties in natural language, but provides no way at all to express structural properties (e.g., the fact that an entity has some attribute). 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, (Bertolino et al., 2002; Fantechi et al., 2002; Richards et al., 2002; Richards and Boettger, 2002)).

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. Indeed, a recent survey among 142 software development organizations (Mich et al., 2004) has found that the market demand for NL-based RE tools is already well developed, and not being adequately served by current commercial offerings.

The CIRCE framework we presented in this paper proposes to address this need by encouraging a rigorous use of NL. By structuring requirements documents in the three layers of 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, 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 such as Entity-Relationship or UML class diagrams, and dynamic models such as 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 like Rational RequisitePro, for design like Rational Rose, and for coding by producing skeleton Java code. As already mentioned, we currently interface with design tools by exporting XMI models. We have also extended this support by providing an integrated Requirements Development Environment embedded in Eclipse, an open platform for tool integration developed by the Eclipse Consortium (Eclipse Consortium, 2003; Ambriola et al., 2003); details on this work are reported in Ambriola et al. (2004).

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 25 teams working on a single server, on different projects, at the same time. Scalability in terms of requirements documents size has been tested up to only 350 requirements in a single project, which is smaller than most industrial projects. Modular and compositional facilities to handle larger requirements bases must be introduced and tested.

Validation of CIRCE has been performed by running three informal pilot studies with industrial partners, and a more controlled study in cooperation with the NASA Independent Verification and Validation (IV&V) facility.

The goal of the three pilot studies was to test the workplace acceptability of CIRCE, together with its robustness, and to collect the subjective impression of industrial users of the tool. The studies were conducted together with industrial partners in Italy and Japan. These were medium-sized companies each employing from 40 to 120 people; two companies worked in the fields of communication technologies, while the third worked on data collection and mining for business and marketing purposes. The companies had different levels of maturity, ranging from totally unstructured development coupled with high craftmanship, to experimental uses of formal methods. We asked our partners to describe the benefits that were provided by CIRCE in the pilot studies. Answers pointed out four major improvements over current development practices:

1. the ability to already examine, validate, and measure models of the software to be produced even at the informal requirements stage;
2. the ability to change, in a consistent way, dozens of models with a single change in the text of the requirements;
3. the ability to get immediate feedback about 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.

The goal of the NASA study, reported more fully in Gervasi and Nuseibeh (2002), was to verify (i) how adaptable CIRCE was to analyze pre-existing industrial requirements, and (ii) how effective it was in analyzing them. Several releases of a fragment of the NASA Software Requirements Specification for the International Space Station were analyzed, and CIRCE was extended both at the parsing stage (by writing new MAS rules) and at the analysis stage (by writing new modelers and application-specific views). The amount of work needed to adapt CIRCE was found to be small when compared to the effort needed to manually analyze and re-analyze the document at each version. Moreover, CIRCE's validation modelers were able to find in the requirements *all* the problems that the manual inspections performed by NASA IV&V had exposed, *plus* a few others, one of which was judged serious and could present a potential security hazard to the crew. The ability to repeat all the validation checks automatically at each change to the requirements (even between releases) was also found to be highly beneficial, as validation could then be performed during requirements writing, when potential solutions to problems exposed by the validation are most readily accessible.

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

1. from our experience in teaching Requirements Engineering and Software Engineering classes, we have found CIRCE to be an effective tool to help develop in students a special sensibility for the *meaning* of the requirements they write, thus promoting better RE education;
2. 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 embody perfectly valid and precise advice that can be followed by humans as well.

In particular, in teaching we have observed and measured, through CIRCE's measurement modelers, a considerable increase of the accuracy and complexity of the requirements produced by our students, following the progressive introduction of more sophisticated views and validation capabilities in CIRCE in the course of several years.

Contrasting the advantages listed above, the main risk in using CIRCE lies in excessive, blind reliance on the tool. CIRCE is meant to act as a supporting companion to a requirements engineer, not to magically substitute for a lack of understanding of the real problems. The wealth of focused information that is presented back to the user is intended to mitigate this risk.

The cost of introducing CIRCE in the work environment can vary, depending on local conditions and on the desired amount of analysis to be performed on the requirements. At one extreme, as tested in the NASA case study cited above, pre-existing requirements can be analyzed by CIRCE as they are, thus imposing no additional burden[20] on the requirements writer. At the other, some training may be needed on the language accepted by CIRCE and on how to interpret the most important views. Our experiences in academic and industrial settings over several years have consistently indicated that the required training can be imparted to third-year CS students or practitioners in eight or ten hours of frontal teaching, distributed over one or two weeks to leave some room for personal experimentation.

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 CIRCE will help promote better requirements engineering practices and advance the state of the art in the field.

Acknowledgments

The authors wish to thank the anonymous referees for their valuable comments and detailed suggestions that led to a significant improvement of the paper.

Notes

1. A synonym is a term that in a given context can be used interchangeably for another; a metonym is a term that denotes one thing but refers to a related thing, e.g. *the White House* standing for the presidential authority in the US.
2. For clarity, we have omitted in this example the tags that would be appended to the various designations.
3. We regard tags as pure labels at this stage; the intended meaning of those used in the examples will be discussed in Section 6. Notice that glossaries need not be written *ex novo* every time: on the contrary, in most applications they are reused for new applications in the same domain, or they can be extracted from standardized, reusable domain ontologies.
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).
6. 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.
7. Non-retractability ensures that requirements and facts deduced by the requirements cannot be negated by a modeler—unless, of course, the text of the requirements themselves is changed.
8. Many other types of communications are considered in CIRCE; the corresponding tuple types are omitted here for brevity.

9. 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.
10. For example, UML models generated by CIRCE can be exported through XMI and imported into other tools such as Rational Rose or ArgoUML.
11. The current implementation of CIRCE has seven representation 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.
12. The user can choose at any time whether new views should be opened in independent windows, or a pre-existing window should be recycled.
13. In fact, other meta-models have been implemented, for different purposes, by writing appropriate parsing rules and modelers. See, for example, (Gervasi and Ambriola, 2003), in which the argumentative structure of literary texts is analyzed and modeled instead.
14. The reader is referred to Ambriola and Gervasi (2006) for details.
15. Most MAS rules dealing with entities use the wildcard tag `/*` (see Section 4.5) rather than `/ENTITY` to support the simpler notation.
16. 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.
17. In the general case, the occurrence of an event is not known beforehand, thus this expression may induce non-determinism in the requirements.
18. In fact, we have used the same technique when we used a definition to establish the meaning of "turning things on/off".
19. 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 that indicates in the glossary the system to be developed. Jackson (1995) has a good discussion on why this approach is preferable to the one based on moods.
20. Except the cost of writing MAS rules to parse the language used in the requirements. However, this is a one-time only operation, that took one of the authors 2 days in the NASA case, so the actual amortized cost is negligible.

References

Abbot, R. 1983. Program design by informal English descriptions. *Communications of the ACM*, 26(11):832–894.

Ambriola, V., Del Carlo, L., and Gervasi, V. 2004. Eclipse as a requirements engineering environment. In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, Edinburgh, Scotland, UK.

Ambriola, V. and Gervasi, V. 1997. Processing natural language requirements. In *Proceedings of the Twelfth International Conference on Automated Software Engineering*, Los Alamitos, IEEE Computer Society Press, pp. 36–45.

Ambriola, V. and Gervasi, V. 1999. Experiences with domain-based parsing of natural language requirements. In G. Fliedl and H.C. Mayr, editors, *Proceedings of the Fourth International Conference on Applications of Natural Language to Information Systems*, number 129 in OCG Schriftenreihe (Lecture Notes), Klagenfurth, Austria, pp. 145–148.

Ambriola, V. and Gervasi, V. 2000. Process metrics for requirements analysis. In *Proceedings of the Seventh European Workshop on Software Process Technology*, pp. 90–95.

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

Ambriola, V. and Gervasi, V. 2006. The CIRCE native meta-model. Technical report, Dipartimento di Informatica, Università di Pisa, (to be published).

Ambriola, V., Gervasi, V., and Del Carlo, L. 2003. The Circe Plugin for Eclipse, http://circe.di.unipi.it/Eclipse.

Balzer, R. 1991. Tolerating inconsistency. In *Proceedings of the Thirteenth International Conference on Software Engineering*, IEEE CS Press, pp. 158–165.

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

Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., and Maccari, A. 2002. Use case description of requirements for product lines. In B. Geppert and K. Schmid, editors, *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL02)*, Essen, Germany, Avaya, Inc., pp. 12-18.

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

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

Buchholz, E., Cyriaks, H., Düsterhöft, A., Mehlan, H., and Thalheim, B. 1995. Applying a natural language dialogue tool for designing databases. In Bouzeghoub, M. and Métais, E., editors, *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB’95)*, Versailles, France,. Association Française des Sciences et Technologies de l’Information et des Systèmes, pp. 119–133.

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

Choi, Y., Rayadurgam, S., and Heimdahl, M.P.E. 2002. Toward automation for model-checking requirements specifications with numeric constraints. *Requirements Engineering Journal*, 7(4).

Colombetti, M., Guida, G., and Somalvico, M. 1983. 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, pp. 163–180.

Dalianis, H. 1992. A method for validating a conceptual model by natural language discourse generation. In *Advanced Information Systems Engineering*, number 593 in LNCS. Springer-Verlag.

Damian, D. and Zowghi, D. 2003. RE challenges in multi-site software development organisations. *Requirements Engineering Journal*, 8(3):149–160.

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

Dubois, E. and Pohl, K. 2002. RE 02: A major step toward a mature requirements engineering community. *IEEE Software*, 20(2):14–15.

Eclipse Consortium. 2003. Eclipse.org main page, http://www.eclipse.org.

Eich, C. 1984. From natural language requirements to good data base definitions—a data base design methodology. In *Proceedings of the First International Conference on Data Engineering*, Los Angeles, USA. IEEE Computer Society, pp. 324–331.

Eshuis, R., Jansen, D.N., and Wieringa, R. 2002. Requirements-level semantics and model checking of object-oriented statecharts. *Requirements Engineering Journal*, 7(4).

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

Fabbrini, F., Fusani, M., Gervasi, V., Gnesi, S., and Ruggieri, S. 1998. Achieving quality in natural language requirements. In *Proceedings of the Eleventh International Software Quality Week*, S. Francisco, California, Software Research Institute.

Fabbrini, F., Fusani, M., Gnesi, S., and Lami, G. 2001. 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.

Fantechi, A., Gnesi, S., Lami, G., and Maccari, A. 2002. Linguistic techniques for use cases analysis. In *Proceedings of the IEEE Joint Internatonal Requirements Engineering Conference (RE02)*, IEEE CS Press: Essen, Germany, pp. 157–164.

Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., and Moreschini, P. 1994. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3):243–263.

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. 1994. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578.

Fuchs, N.E. and Schwertel, U. 2003. Reasoning in Attempto controlled English. In *Proceedings of the International Workshop on Principles and Practice of Semantic Web Reasoning*, number 2901 in Lecture Notes in Computer Science. Springer-Verlag.

Gabbay, D. and Hunter, A. 1991. Making inconsistency respectable: A logical framework for inconsistency in reasoning, Part 1—a position paper. In P. Jorrand and J. Kelemen, editors, *Proceedings of Fundamentals* *of Artificial Intelligence Research*, Vol. 535 of *Lecture Notes in Computer Science*, Springer-Verlag: New York, pp. 19–32.

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

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

Gervasi, V. 2001b. Synthesizing ASMs from natural language requirements. In R. Moreno-Díaz and A. Quesada-Arencibia, editors, *Proceedings of the Eighth EUROCAST Workshop on Abstract State Machines*, Las Palmas, Spain, Universidad de Las Palmas, pp. 212–215.

Gervasi, V. 2002. Integrating Circe and Lyee. Circe Project internal report 19/02.

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

Gervasi, V. and Nuseibeh, B. 2000. Lightweight validation of natural language requirements: A case study. In *Proceedings of the Fourth International Conference on Requirements Engineering*, pp. 140–148.

Gervasi, V. and Nuseibeh, B. 2002. Lightweight validation of natural language requirements. *Software–Practice and Experience*, 32(2):113–133.

Goldin, L. and Berry, D.M. 1997. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engineering*, 4(4):375–412.

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

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

Jackson, M. 2002. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley.

Johnson, J., Boucher, K.D., Connors, K., and Robinson, J. 2001. Project management: The criteria for success. *Software Magazine*.

Macias, B. and Pulman, S.G. 1995. A method for controlling the production of specifications in natural language. *The Computer Journal*, 38(4):310–318.

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

Mich, L. 1996. 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.

Mich, L., Franch, M., and Novi Inverardi, P. 2004. Market research for requirements analysis using linguistic tools. *Requirements Engineering Journal*, 9:40–56.

Mich, L., Mylopoulos, J., and Zeni, N. 2002. 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.

Neill, C.J. and Laplante, P.A. 2003. Requirements engineering: The state of the practice. *IEEE Software*, 20(6):40–45.

Nuseibeh, B. and Easterbrook, S. 2000. Requirements engineering: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, ACM Press.

Object Management Group. 2002. *OMG-XML Metadata Interchange (XMI) Specification, Vol. 1.2, http://www.omg.org/technology/documents/formal/xmi.htm.*

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

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

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

Richards, D. and Boettger, K. 2002. 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, BCS.

Richards, D., Boettger, K., and Aguilera, O. 2002. A controlled language to assist conversion of use case descriptions. In B. McKay and J.K. Slaney, editors, *Proceedings of the Fifteenth Australian Joint Conference on Artificial Intelligence*, Vol. 2557 of *Lecture Notes in Computer Science*, Springer-Verlag: Canberra, Australia, pp. 1–11.

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

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

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

Schwitter, R. 2002. English as a formal specification language. In *Proceedings of the Thirteenth International Workshop on Database and Expert Systems Applications*, Aix-en-Provence, France, pp. 228–232.

Schwitter, R., Ljungberg, A., and Hood, D. 2003. ECOLE: A lookahead editor for a controlled language. In *Proceedings of EAMT-CLAW03 (joint conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop)*, Dublin, Ireland, pp. 141–150.

Sukkarieh, J. and Pulman, S. 1999. Computer processable english and McLogic. In H. Bunt et al., editor, *Proceedings of the Third International Workshop on Computational Semantics*, Tilburg, pp. 367–380.

van Lamsweerde, A. 2000. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, ACM Press.

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