Supporting Multiple Views on Requirements

Vincenzo Ambriola
+39 050 887251
ambriola@di.unipi.it

Vincenzo Gervasi
+39 050 887273
gervasi@di.unipi.it

Dipartimento di Informatica, Università di Pisa
Corso Italia, 40 — I–56126 Pisa, Italy

ABSTRACT

The quality of a software product is greatly influenced by the quality of its requirements. In order to obtain high quality requirements, it is important to take advantage of all the available skills related to a project, by allowing all the stakeholders to participate actively in the requirements writing process.

The differences in knowledge, expertise, background, and goals among the stakeholders, however, may make communication among them difficult: some form of automatic support is needed to make the process practical.

In this paper we consider the issue of providing different multiple views on the requirements to different stakeholders, varying in depth, focus, and representation. Basic requirements for an environment supporting this capability are discussed, and a reference architecture satisfying those requirements is given. We also present a web-based prototype implementation of the architecture, concluding with some experience on its use.

Keywords
Requirements Engineering, Tools and Environments.

1 INTRODUCTION

According to [11], the quality of a product can be defined as its ability to satisfy the implied or expressed needs of those affected by its existence. Thus, quality-seeking approaches must take into account a more or less direct participation of all the persons and organizations that hold legitimate interests in the system development and in its operations, collectively called the *stakeholders* in a particular product. When considering software products, the stakeholders generally include at least a customer and a developer, but may include other roles (e.g., users of the system, the customer's customers, regulating bodies like environment protection agencies or government, etc.). Also, certain products could lack a specific customer, as in the case of packaged software, or even a specific developer, as in the case of the preparation of a call for tenders.

In all these cases, fruitful cooperation among the stakeholders is probably the best means to exploit available knowledge and competence without long, costly, error-prone, and often painful knowledge transfers. A number of works ([4, 5, 6, 8, 13] among others) in the last decade have analyzed the effects of user participation in requirements analysis, concluding that projects with more direct participation had a greater success ratio than those with weak participation. We advocate that a similar relation holds for other kinds of stakeholders.

Calling for wider participation, however, poses several problems. The most difficult to solve lies in the different cultural backgrounds of the different stakeholders, often made worse by different work habits and environments, different languages and notations, different areas of competence and overall goals. In order to make wider participation in requirements writing practical and economically viable, these obstacles to communication among stakeholders should be removed or at least reduced as far as possible. The problem of reducing this "communication impedance" among people is a very general one. We concentrate here on the much more restricted case of communicating requirements for a software system, and tackle this problem by designing supporting environments to reduce the translation burden imposed to humans.

In Section 2 we introduce an original framework describing basic activities in requirements writing, while Section 3 focuses on the issue of providing multiple views on the requirements. We present some requirement for an environment offering multiple views support, and in Section 4 an architectural template is introduced and is shown to satisfy the proposed requirements. Section 5 is devoted to presenting our prototype implementation of the architecture, a multiple views supporting environment called Circe, and is followed by an account of some case study. A survey of related works, some conclusions, and ideas for future work complete the paper.

2 A CONCEPTUAL FRAMEWORK

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

Linguistic Properties of Requirements

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

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

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

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

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

Basic Activities on Requirements

Of a requirements engineering activity, we observe only the repeated exchange of requirement proposals among the stakeholders, leading in time to (1) a set of requirements acceptable to all the stakeholders, and (2) a set of residual conflicts between stakeholders, together with their rationales. This exchange can be decomposed into two activities:

- **Expression.** Requirements are at first expressed by a stakeholder. This activity associates a syntactic and lexical form to the requirements semantics (i.e., what the stakeholder wants to express). The syntactic form provides a frame for the semantic content, while the lexical form links the requirement to some set of agreed-upon concepts. After expression, requirements have a physical form, and are thus amenable to transmission and processing. In particular, they can be communicated to other stakeholders.
- **Comprehension.** Requirements are then understood by other stakeholders. In our framework, understanding refers to the operation of extracting the semantic content of a requirements from its syntactic and lexical form, by using some base knowledge[1] about the language used to express the requirement, and about the problem and the domain of the system being designed.

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

Communication, however, is only a step in our framework. The success of the mere communication of requirements *per se* is not sufficient to ensure the effective participation of all the stakeholders. In fact, comprehension leads to three other activities:

- **Modeling.** Different stakeholders tend to reason about the system to design in terms of different abstractions, varying in depth, detail and focus, and according to their goals and interests. Different abstractions correspond to different models, so the ability to build models is essential for an effective participation of the stakeholders.
- **Validation.** When considering a candidate requirement proposed by another stakeholder, each stakeholder implicitly performs several kind of validations, e.g., internal consistency of the proposal, or compatibility with the stakeholder's goals. The outcome of this validation step determines the acceptability of a proposed requirement, or exposes conflicts among the different points of view.
- **Metrication.** A number of technical and management decisions depend on the ready availability of different measures on the requirements, on the system described by the requirements, and on the evolution of the requirements in time. A stakeholder

[1] Note that we are *not* asking here that this knowledge be common to all the stakeholders.

must be in a position to measure or estimate relevant attributes in order to take informed decisions during his or her work.

It is important to remark that all these activities are *not* sequential steps of a well-defined process. In the vast majority of cases, they occur as mental processes of the stakeholders — that are often unconscious. The only synchronization points among the stakeholders that we are interested in are the communication of proposed requirements, amendments, rejections, counter-proposals, etc. In fact, the issues related to multiple views support (detailed in the next section) are essentially orthogonal to any specific process.

3 THE MULTIPLE VIEWS PROBLEM

The particular problem we focus on in this paper, with reference to the framework we presented above, is the issue of presenting different multiple views (MV) on the requirements to different stakeholders, according to their knowledge, background, goals and to their role in the development of a software product. While this is essential when different stakeholders do not understand each other's "language", it is also of great benefit in simple cases where interaction is very limited. In fact, working with more focused views greatly simplifies the correct definition of particular aspects of the system to design.

We propose a solution to this problem based on the introduction of an environment supporting the basic activities of our framework.

Requirements for MV Environments

We identify the following requirements for an environment supporting multiple views on the requirements:

1. The environment must allow the expression of models of the system to design and of its environment, in a way that is independent from a particular syntactic form;
2. The environment must be able to store and retrieve expressed models;
3. The environment must be able to synthesize models of the system to design and of its environment, using stored information, in a way that is independent from a particular syntactic form;
4. The environment must maintain the lexical content of the models, and should preserve pragmatic aspects as far as possible;
5. The environment must be able to perform a formal validation of stored models, and must support the verification of properties of such models;
6. The environment must be able to extract metric data about the system to design and about the requirements process itself;

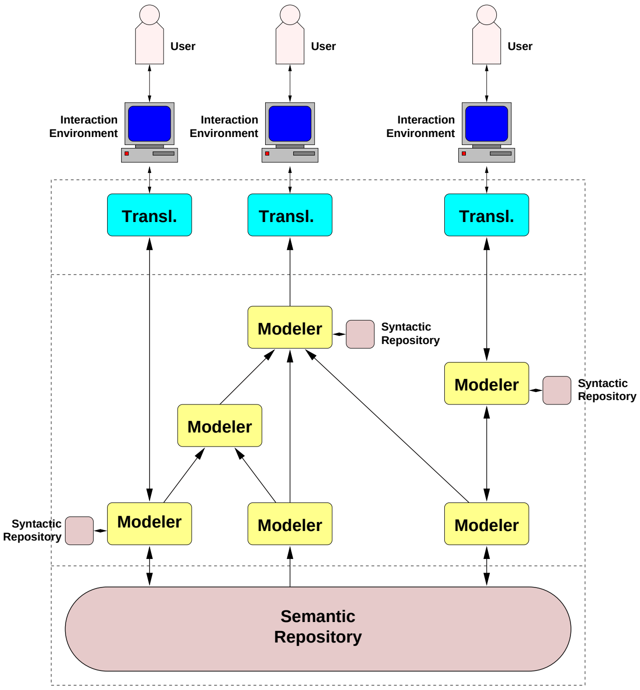


Figure 1: Reference architecture for a MV supporting environment.

7. The environment must provide suitable user-level representations (views) for models, validation reports and metric data;
8. The environment must allow the easy integration of arbitrary models and representations.

We consider these requirements as necessary, but not sufficient, conditions for an effective support. In particular, environments *not* satisfying one or more of these requirements will present significant shortcomings when used in a MV context. See Section 7 for examples of such systems. Other requirements can be given on important issues like navigability, traceability, user interface, etc., but we do not discuss these issues in the present work.

4 A REFERENCE ARCHITECTURE

In this section, we introduce a reference architecture for MV supporting environments; we also show how the reference architecture fulfills the requirements we stated in the previous section. The reference architecture is meant as a *template* only; see Section 4 for a discussion of its role and Section 5 for an instantiation of the template.

Overview

As can be seen in Figure 1, our architecture is based on four kinds of components, tied together by an interconnection structure. At the basis we have a central knowledge repository, that we call *semantic repository*, storing the semantic information input as requirements by the users of the environment. The semantic repository stores very basic semantic information fragments, of a fine granularity; we call such fragments *atomic requirements.* Examples of atomic requirements are a single arc in a data flow diagram, or the existence of a single attribute for an object in an entity-relationships diagram. It is worthwhile to stress here that the semantic repository does *not* necessarily contain a correct, consistent set of requirements. On the contrary, it stores every single fragment of information input by every user of the environment, leaving correctness issues to other components.

Model building, validation and metrication are performed by specialized components called (with some bias) *modelers*, arranged in a network structure. These components perform two basic functions in our architecture. Their main purpose is the synthesis of a model starting from atomic requirements and fragments of lower-level models. This can be seen as the processing of information flowing upward in Figure 1. In the case of *editable* models, a modeler can also decompose models input by the user into atomic requirements, to be stored in the semantic repository, so that knowledge acquired via a model can be profitably used in building different models. In this case, too, a modeler can use lower-level modelers to gradually decompose a complex model into atomic requirements. This corresponds to the processing of information flowing downward in Figure 1.

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

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

Other components called *translators* constitute the third kind of components of our architecture. Translators are in charge for building an effective representation of a model (that, up to this point, exists only in abstract form) and of parsing representations input by the user. Users can only introduce or examine models via one of their representations, as produced by a translator. The actual user interaction, that can be as simple as the reading of some text, or as complex as the editing of a complex graphical diagram, takes place in an *interaction environment* (constituting our fourth kind of components), potentially different from user to user.

Does the Architecture Satisfy Its Requirements? In this section we show how the proposed architecture satisfies the requirements we identified in Section 3.

1. **Expressing models.** Thanks to the presence of the various translators and modelers, each user can express the models he or she is interested in, and use the expressive means that are more adequate to the problem at hand and to the user's knowledge and habits. To assure that this requirement is satisfied three conditions must be met:
   (a) The environment must include a modeler for the model to be expressed (and for any lower-level model upon which it is based);
   (b) The environment must include a suitable translator, accepting as input the desired representation, and producing as output the corresponding model;
   (c) The interaction environment must allow the user to edit the chosen representation.
2. **Storing and retrieving models.** In our architecture, the semantic content of models is stored in the centralized semantic repository, so that it can be used by other modelers. The syntactic and pragmatic contents are stored in the specialized syntactic repositories, while the lexical content is present in both repositories.
3. **Model synthesis.** The opportunity for a modeler to synthesize a model is ensured, in our architecture, by its ability to access both the semantic repository data and the models built by other modelers. There are no architectural constraints on the representations that can be obtained, as long as
   (a) The environment includes a modeler for the desired model (and for any lower-level model upon which it is based);
   (b) The environment includes a suitable translator, accepting as input the given model, and producing as output the desired representation;
   (c) The interaction environment is capable of visualizing the chosen representation;
   (d) The semantic repository contains enough information to synthesize the desired model.
4. **Preservation of lexical and pragmatic aspects.** Since the relevant information is stored in the semantic and syntactic repositories, modelers can maintain the lexical content of the requirements, and translators can reproduce or map the pragmatic aspects as appropriate.

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

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

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

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

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

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

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

Discussion

The architecture we presented in this section is a *reference* architecture, meaning that it can be (and has been) implemented in totally different ways. Some typical implementation styles include:

- Active-deductive database. In this case the repositories constitute the extensional data base, while modelers and translators constitute the intensional data base. Extendibility can be obtained by considering each modeler as a separate theory of an heterogeneous deductive data base; see [3] for a detailed proposal.

- Information processing network. The contents of the semantic repository are processed by the various modelers and translators acting as progressive filters. This is the style adopted by our prototype described in Section 5.

- Agent system. Each modeler and translator acts as an autonomous agent working on a shared blackboard containing representations of the various repositories. Coordination is needed to ensure an appropriate scheduling of the agents.

The reference architecture can also be used as a framework for classification of similar systems, as we have done in Section 7.

5 Circe: A PROTOTYPE MV SUPPORTING ENVIRONMENT

In this section we present Circe, a prototype MV supporting environment we designed and implemented. Circe is an instance of the reference architecture presented in Section 4, and as such it offers complete multiple-views support. Circe has been used with promising results in a number of projects, both academic and in cooperation with industrial partners, on projects of small to medium dimension (up to 250 requirements). In most of these cases, users of Circe deemed MV support to be a useful addition to their RE processes and showed satisfaction with the performance of the environment.

In order to instantiate our reference architecture, we have to define our components — the structure and contents of the semantic repository, the number and function of the various modelers and translators, the interaction environment —, and our connectors — the interconnection structure among all the different components composing our architecture. In the following, we describe in turn each of these aspects.

The Repositories

Circe's semantic repository is implemented as a flat typed-tuples space, where each tuple has the form $\langle\ tid, \mathsf{type}, arg_1, \ldots, arg_n\ \rangle$. The type indicates the kind of semantic information embodied by the tuple, and a set of arbitrary parameters specifies the semantic content. Usually, parameters are either terms from the lexical part of some requirement or references to other tuples, via their tuple identifier *tid*. In this way the repository, although stored as a flat set, can actually maintain information shaped in more complex structures (e.g., trees or graphs). The semantic repository can be queried via basic queries based on the usual regular expression syntax, or by using a more sophisticated matching operator to select entire substructures. Figure 2(b) constitutes an example of the contents of our semantic repository.

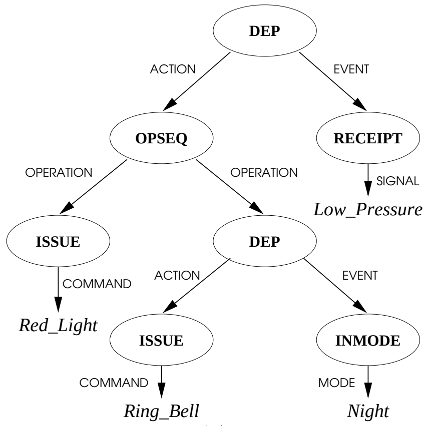
[2] This approach has additional advantages on compositionality grounds; for example, in our prototype a higher-level modeler builds a model on top of various validation modelers that prioritises all problems found according to their seriousness.

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

(a)

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

(b)



(c)

Figure 2: A requirement (a), the corresponding tuples (b), their structure (c).

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

In the current version of Circe, all repositories are kept under formal control (through the classic RCS system). It is thus possible to trace back the entire evolution of the requirements, for analytical, statistical, or metric purposes. This is made possible in our architecture by considering RCS repositories as syntactic repositories of a specialized modeler.

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

Each modeler is a triple $\langle input, output, behaviour\rangle$ where *input* declares which tuples are expected as input (by listing or giving a pattern of their types), *output* declares which tuples are produced as output (by listing their schemas), and *behaviour* describes the computation needed to produce the output tuples. Although the behaviour part can be arbitrary, we have found AWK scripts to be a practical and reasonably efficient way of processing textually-encoded tuples.

As an example, Figure 3 shows a very simple modeler providing a "black box" model of a system (showing only the inputs and outputs of the system). The BlackBox modeler's behaviour specifies that, whenever an issue or receipt tuple is found in its input, the corresponding arguments are collected in the `in` and `out` variables. When all the input tuples have been processed, two new tuples (hasin and hasout) are produced, listing the inputs and outputs of the system. The model computed by the BlackBox modeler from the repository of Figure 2(b) is also shown in Figure 3.

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

The Translators
Circe currently implements almost 30 translators, that are in charge for building concrete representations of the abstract models produced by the modelers; translators for editable representations also provide parsing capabilities for a given representation. A translator is a triple $\langle input, output, behaviour\rangle$ where *input* is the same as for modelers, while *behaviour* describes how the concrete representation is computed, and *output* declares the symbolic name of the representation (for user display purposes). A sample translator for our BlackBox model is shown in Figure 4.

A translator that deserves special mention is our natural language parser Cico, that has been described elsewhere [1, 2]. It can parse NL requirements according to a given set of rules, and produce tuples that are then injected in the semantic repository. In Figure 2, the translation from the NL text in (a) to the tuple space in (b) is an actual example of Cico's output.

The Interaction Environment
Circe uses as interaction environment one of the environments that are most familiar to users, namely that of the World Wide Web. Representations output by the translators are converted by the interaction environment into HTML, Java and JavaScript. This choice maximizes the availability of the interaction environment: a standard Web browser is enough to access and

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

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

Figure 3: The BlackBox modeler (left) and the model it computes on the repository of Figure 2(b) (right).

```
BlackBox translator
#INPUT: (hasin|hasout)
#OUTPUT: "Black-box view"
hasin   { for (i in $*)
            mkarc(nil,"System",i) }
hasout  { for (i in $*)
            mkarc("System",nil,i) }
```

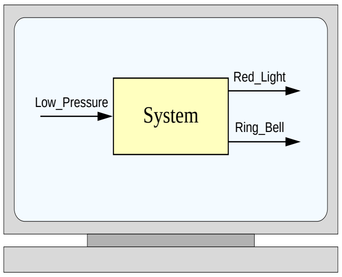


Figure 4: The BlackBox translator (left) and the representation it produces from the BlackBox model of Figure 3.

work profitably with Circe. This also simplifies the work of translators, that can rely on the layout capabilities of HTML and on the availability of standard Java or Javascript components to generate the respective representations.

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

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

The Interconnection Structure

The interconnection mechanism in Circe is quite simple, as it is implemented via standard UNIX pipes. The interconnection structure, on the other hand, is defined dynamically for each computation by a scheduler inside Circe's kernel.

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

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

A Sample Scenario

Figure 5 shows a work session in the interaction environment provided by Circe. On the top, an editable

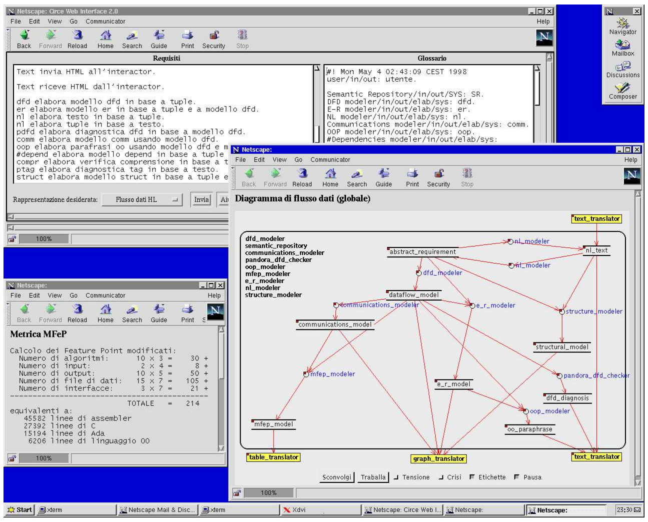


Figure 5: A session with CIRCE. Three views on the same requirements are shown; from the top in clockwise order: NL requirements, data flow diagram, functional metrics.

textual representation of the system being designed allows the user to write NL requirements. On the bottom right corner, the corresponding data flow model is shown, as built by the DFD modeler from the information introduced into the semantic repository by the NL translator CICO.

In a typical case, some non-technical stakeholder (e.g., a customer) could write a first NL requirements document, thus providing an initial sketch of the system to design. Another stakeholder, more technically educated (e.g., a developer), could look at the corresponding data flow diagram, synthesized by a chain of modelers from the original requirements, and spot some missing flow or computation. The developer could copy and paste this view into an e-mail, discuss the issue with a colleague, and then simply add the missing element by editing the diagram. The translator for data flow diagrams would then parse the graph underlying the representation, and produce high-level tuples corresponding to the added elements. This time, a chain of modelers would be run to decompose the high-level tuples into low-level ones to inject in the repository.

The change would finally be propagated to the first stakeholder's view of the system, by adding the corresponding NL statement (synthesized from low-level tuples by applying CICO's parsing rules in the dual direction) to his or her document. The customer could then easily confirm whether the developer's understanding leading to the modification was correct or not with respect to his or her own point of view.

Meanwhile, a third stakeholder (e.g., a project manager) could be interested only in the functional score of the specification (shown in the window in the bottom left corner in Figure 5), to ensure that the projected cost for the system does not exceed allocated budget. Requirements evolution could continue in a similar way, exploiting the combined work and competencies of all the participants, until a stable set of requirements is obtained and conflicts have been solved by mutual agreement or by managerial decisions.

Incidentally, the system being designed in the session captured in Figure 5 is CIRCE itself.

6 CASE STUDIES

We have conducted a number of case studies with CIRCE in order to evaluate the effects of its introduction in real requirements processes.

In one case, we used the tool for the partial validation of a fragment of NASA's specification of the Node Control Software of the International Space Station [15]. The choice of this particular document was appealing because we assumed it to be of high quality (being the 12th release of those requirements), and because parts of it had already been analyzed using different techniques, in related studies [7, 20]. Our study focused on the Cabin Pressure Monitoring function, part of the Environmental Control function. The requirements describing cabin pressure monitoring were input into CIRCE, translated by CICO into atomic requirements, and processed by our modelers. Various validation modelers uncovered a total of over 10 formal errors in the original specification. Most of these were also independently revealed by NASA's verification & validation process. Moreover, the ability to directly produce focused views from the requirements (e.g., events-actions tables) allowed us to identify two subtler substantial errors that were not identified by the inspections-based process used by NASA. In effect, our multiple views approach allowed us to inspect in insulation a number of well-focused documents, each detailing a single facet of the requirements. On the contrary, the traditional inspection process used by NASA required that the entire requirements document be inspected in its entirety. Full details on this case study are reported in [10].

Similar results were obtained in other case studies, both conducted by the authors and by industrial third parties. The experiences so far confirm our hypothesis that providing multiple views on requirements is an effective way to improve the stakeholders' ability to contribute to their full potential to a requirements process.

7 RELATED WORK

The idea of offering some automatic support for the difficult processes of requirements elicitation and validation is not new; systems similar under certain respects to the one we described exist since at least 10 years. Some confidence on the value of our framework can be gained by noticing that in most cases the architecture of such systems maps quite nicely onto our reference architecture, and furthermore, that the absence in a tool of some kind of components of the reference architecture often make it less suitable to usage by different stakeholders.

One of such systems is the “Requirements Apprentice”, by Reubenstein and Waters [18, 19]. In contrast with other solutions, the Requirements Apprentice does not stress the fact that requirements must be expressed in a formal language to start with, but rather tries to facilitate the transition from informal to formal requirements by using a natural language-like syntax. This is comparable in purpose to the role of the NL translator Cico in Circe. Also, the system uses some knowledge about the application domains (in the form of so-called clichés) to validate the requirements.

Internally, the Requirements Apprentice uses a standard frame-based knowledge repository both for clichés and for requirements; this repository maps onto our semantic repository. The system does not explicitly discriminate between syntactic and semantic aspects of the requirements, i.e., it lacks distinct notions of modeler and translator; this is reflected by the small (and fixed) number of representations available. Operating on the repository, a hybrid reasoning system called Cake [9] can detect contradiction and incompleteness, disambiguate ambiguous sentences, and provide natural language output, thus producing actual requirements documents. Cake would be just a particular case of a validation modeler in our framework, while its other capabilities can easily be mapped on the corresponding modelers.

In a more recent work, Maiden presents the AIR Toolkit [14]. Among other functions, the toolkit features two tools called “Requirement Refiner” and “Requirement Critic”. The first tool provides a sort of structured editor for natural language requirements, that forces the user to write requirements according to one of a couple of hundred of pre-defined “problem abstractions” (similar to the clichés we cited above); the second tool performs a validation on such requirements via an interactive dialogue with the user. While no detail is given about the internal architecture of the system, we can easily identify the Requirement Refiner as the composition of a simple NL modeler with a more complex structured editing translator, and the Requirement Critic as a validation modeler coupled with an interactive translator.

Johnson et al., in [12], discuss the approach taken to requirements presentation in their tool ARIES. Requirements and domain description statements are entered into the system by using a declarative language, and stored in an internal knowledge base (again, the knowledge base maps onto our semantic repository). ARIES can then present this knowledge to the user either in textual form (by using the GIST explainer [21]), or in graphical, diagrammatic form. Their work is original in that the diagrammatic form can be directly manipulated by the user by applying a number of *stereotypical operations* (ARIES provides over 180 operations) that propagate their semantics to the underlying knowledge stored in the knowledge base. This kind of structured editing (this time on the diagrammatic form rather than on the textual form) can be easily modeled in our framework with an appropriate translator. ARIES uses the GIST explainer as we use the unparsing capabilities of Cico, and uses a declarative language whereas we allow input through different representations.

A different approach has been taken by The Viewer [16], a system designed to test the limits of distributed requirements engineering. In contrast with our framework, where a centralized repository is accessible by every modeler, in The Viewer each modeler (embodying a viewpoint [17]) maintains its own completely insulated repository; both modeler and repository are implemented as SmallTalk objects. This design offers great flexibility, but the price to pay is the need to explicitly define communication protocols for *each pair* of modelers, whereas in Circe only a single protocol has to be defined. In The Viewer, each viewpoint integrates a modeler and a translator; however, thanks to the object-oriented implementation, the same translator can be easily reused for several viewpoints, effectively making it a separate component. The Viewer also devotes much attention to the issue of inconsistency detection and to process support, areas where Circe is currently lacking.

Many more systems have been proposed than we can cite here; the observation that most of them can be mapped to the right combination of components in our reference architecture makes us confident that we have attained sufficient generality.

8 CONCLUSIONS

Improving the stakeholders' ability to cooperate through multiple different views on the requirements seems a promising approach, but it cannot deliver without a flexible and practical supporting environment. In this paper we have identified some of the basic requirements (those related to multiple views support) that a supporting environment should satisfy in order to allow all the stakeholders in a project to participate profitably in the requirements writing and validation phases. We also proposed a reference architecture and showed how our architecture satisfies those requirements.

An implementation of the architecture, a supporting environment called Circe, has been developed by the authors both as a proof-of-concept and as a prototypical tool, allowing us to observe the consequences of its use on the quality of the resulting requirements. We have run several experiments, both in an academic context and in collaboration with industrial partners. Although a formal assessment of the impact of CIRCE's introduction has not yet been conducted, users have shown satisfaction with the ease of interaction and added control on their requirements that our environment has brought to them.

Future works on CIRCE will surely include better and more diverse input methods, together with more user-friendly navigation among different models and representations. Other important issues are those of inconsistency management and of concurrent work.

Within their limitations, we believe that our experiences with CIRCE provide a good indication of the value and practical benefits of our approach.

REFERENCES

[1] V. Ambriola and V. Gervasi. An environment for cooperative construction of natural-language requirement bases. In *Proc. of the 8th Conference on Software Engineering Environments*, pages 124–130. IEEE Computer Society Press, Mar. 1997.

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

[3] E. Bertino, B. Catania, V. Gervasi, and A. Raffaetà. A logical approach to cooperative information systems. *Journal of Logic Programming*, 43(1), Feb. 2000.

[4] P. G. Brown. QFD: Echoing the voice of the customer. *AT&T Technical Journal*, 70(3):18–32, Mar.-Apr. 1991.

[5] A. Cucchiarelli et al. Supporting user-analyst interaction in functional requirements elicitation. In *Proc. of the 1st Asia-Pacific Software Engineering Conference*, pages 114–123, Los Alamitos, California, Dec. 1994. IEEE CS Press.

[6] J. M. Drake, W. W. Xie, W. T. Tsai, and I. A. Zaulkernan. Approach and case study of requirement analysis where end users take an active role. In E. Straub, editor, *Proc. of the 15th International Conference on Software Engineering*, pages 177–186, Baltimore, MY, May 1993. IEEE Computer Society Press.

[7] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirement modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, Jan. 1998.

[8] K. El Emam, S. Quintin, and N. H. Madhavji. User participation in the requirements engineering process: an empirical study. *Requirements Engineering Journal*, 1(1), 1996.

[9] Y. A. Feldman and C. Rich. Principles of knowledge representation and reasoning in the FRAPPE system. In *Proc. of the 6th Israeli Symposium on Artificial Intelligence*, Dec. 1989.

[10] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements: a case study. (Submitted for publication), 1999.

[11] ISO/IEC 9126. *Information technology – Software product evaluation – Quality characteristics and guidelines for their use*. International Organization for Standardization, 1991. 1st edition.

[12] W. L. Johnson, M. S. Feather, and D. R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, Oct. 1992.

[13] M. Keil and E. Carmel. Customer-developer links in software development. *Communications of the ACM*, 38(5):33–44, May 1995.

[14] N. A. M. Maiden. Reuse-oriented requirements engineering in NATURE. *ACM SIGSOFT Software Engineering Notes*, 20(3), July 1995.

[15] NASA/Boeing. Software requirements specification for the NCS MDM CSCI. International Space Station Document S684-10174, Apr. 1997.

[16] B. Nuseibeh and A. Finkelstein. Viewpoints: A vehicle for method and tool integration. In *Proc. of the Int. Workshop on CASE (CASE'92)*, pages 50–60, Los Alamitos, California, July 1992. IEEE CS Press.

[17] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, Oct. 1994.

[18] H. B. Reubenstein and R. C. Waters. The requirements apprentice: An initial scenario. In S. Greenspan, editor, *Proc. of the 5th International Workshop on Software Specification and Design*, pages 211–218, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[19] H. B. Reubenstein and R. C. Waters. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, Mar. 1991.

[20] A. Russo, B. Nuseibeh, and J. Kramer. Restructuring requirements specifications. *IEE Proceedings: Software*, 1999.

[21] W. R. Swartout. GIST English generator. In D. Waltz, editor, *Proc. of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982. AAAI Press.