From User Needs to Requirements

An experience in tool-supported refinement of requirements

Vincenzo Gervasi
Dipartimento di Informatica, Università di Pisa

January 10, 2000

Abstract

A vague and incomplete description from the customer of what a software system should do is often not sufficient to obtain a complete and consistent specification for the system. A requirements document should be as formal and detailed as possible, but not so formal that hinders comprehension by the customer, nor so detailed that facts the requirements engineer needs are obfuscated by the wealth of information.

As a solution to this problem, we propose the adoption of tools and environments supporting the requirements engineering process. In this case study, we refine the problem description document for the Light Control System into a more precise document by using our environment CIRCE. Automatic modeling and validation allow us to identify a number of problems in the original document, and to propose solutions, with very little human effort and a strong guarantee of repeatability.

1 Introduction

Requirements engineering is the process of refining a description of some *user needs* (or *real-world goals*, as in [17]) into a *precise specification* of a software system satisfying those needs.

In this process, an initial document is gradually transformed into progressively more refined documents, that assume a variety of names: User Requirements, Detailed Requirements, Engineering Requirements, High-Level Specification, etc. Two key factors driving the refinement are *concretization* and *formalization*. Concretization consists in providing information that is not present in the original document (i.e., filling in details), while formalization consists in re-stating the same information provided in the original document in a less ambiguous way (i.e., writing in a language with a defined formal semantics).

In a classical requirements elicitation process, the customer provides the information that is needed for concretization, while the requirements engineer is

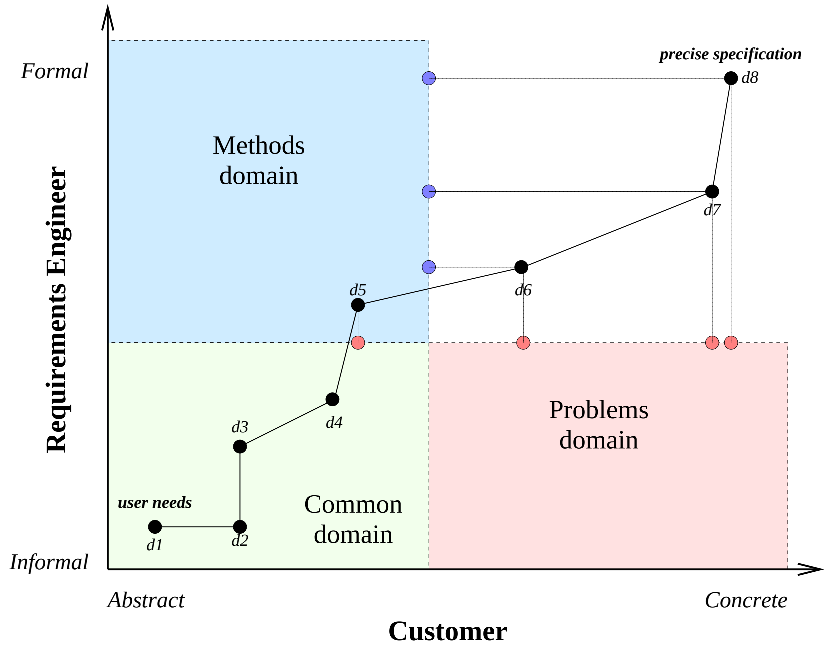

Figure 1: Concreteness and formality in requirements engineering processes.

responsible for formalization.[1] The diagram in Figure 1 shows an example of how user needs are refined into a specification through a sequence of concretization and formalization steps. In the space described by the two dimensions of formality and concreteness, we can identify distinct domains. The knowledge of formal methods and tools, of the semantics of modeling and specification languages, and of elicitation and analysis techniques fall in the *Methods domain*. Typically, a competent requirements engineer has a thorough knowledge in this domain. On the other hand, the knowledge about a particular business domain, about a certain problem and its environment, and about constraining factors on its possible solutions, pertain to the *Problems domain*. Typically, the customer for a software system has in-depth knowledge about his or her problem.

In many cases, the requirements engineer will be able to understand the problem at hand, if only in abstract terms. In the same way, the customer will often master the methods and languages used, up to a certain level of sophistication. The intersection between the Methods domain and the Problems domain, that we call the *Common domain*, represents this area where the requirements engineer and the customer share knowledge and expertise, and can thus communicate freely with little room for misunderstanding. Unfortunately, the ultimate goal of the process is to build a precise specification of the system

[1] In most case, the requirements engineer also has the organizational responsability of pushing for concretization, especially when the added information is needed to discriminate ambiguous cases and thus to attain better formalization.

to design, and this kind of document needs to be both highly formal and very concrete. Thus, in Figure 1, the precise specification is placed in the top right position, and lies outside the domains we described above. As a consequence, while in the initial stages of the requirements engineering process both participants have good control on the documents produced (e.g., the initial "user needs" and subsequent refinements $d_2 \dots d_4$), in later stages they only have control on the *projections* of the documents on their area of expertise (e.g., d_6, d_7, and the final specification).

The phenomenon we described is the root cause of the infamous *communication gap* between the customer and the requirements engineer to which so many project failures have been ascribed. In fact, as soon as the document becomes too formal for the customer to understand, or too detailed for the developer to follow, participants in the requirements engineering process can only reason on the respective projections — a situation which paves the way for the introduction of errors that are undetectable by both parts.

It is thus particularly important to provide some form of support to the concretization and formalization activities leading to the final specification. In this paper, we present an application in this direction of our requirements engineering environment CIRCE. CIRCE provides support for writing "good" natural language requirements, while at the same time interpreting them to extract more formal models that can be inspected and validated against a set of defined properties. We use CIRCE to refine a Problem Description document into a set of precise requirements (still in natural language). We then analyze the models synthesized by CIRCE from the requirements to identify inconsistencies, ambiguities, and areas in need of better explanation. Our findings are then validated through customer's feedback.

It is important to remark that in our exercise we will *not* produce a complete, formal *specification* of the LCS. Rather, we will produce a set of detailed *requirements* from which a specification can be derived, together with a guarantee that our requirements[2] satisfy a certain set of desired properties.

In Section 2 we introduce our requirements engineering environment CIRCE, by describing its architecture and some of its components, together with an example of what kind of information CIRCE can extract from natural language requirements. In Section 3 we describe our methodological approach to the LCS case study, that is presented in detail in Section 4. We then provide an evaluation of the effectiveness of our approach based on the feedback from the customer, and outline the work that would be needed to move from our refined problem description document to a complete specification. In Section 7 we reflect on the lessons we learned from the case study; a short survey of related work concludes the paper.

[2] Or, more precisely, both our requirements document and the system described by those requirements.

2 Circe: a requirements engineering environment

Circe [1] is a Web-based environment supporting requirements writing and analysis. While Circe is a powerful environment supporting many requirements engineering chores (e.g., tracing of requirements, building quality models, querying requirements documents, etc.), in this paper we use only its *multi-views* capabilities, i.e., the ability of providing different views on the requirements, differing in scope, focus, language, and notation.

2.1 Architecture of the system

Circe is built around a component-based architecture. A number of modules called *projectors* are in charge of parsing input requirements and of synthesizing new representations for the requirements. Other components called *modelers* analyze parsed requirements in order to inject their semantic content into a common repository, from which new models of the system described by the requirements can be extracted (and possibly presented to the user via a projector).

In a typical application, a user of Circe (e.g., the customer) inputs natural language requirements by editing a textual document. Circe parses the requirements and stores the corresponding parse trees in a common repository, providing some feedback about the outcome of the parsing process to the user. Later, another user working on the same project (e.g., the developer) may ask for a data-flow oriented view on the system. Circe would then call upon its set of modelers that would cooperate in building the required model from information stored in the common repository. The resulting abstract data-flow model is then sent to a graphical projector to obtain a visible representation, that the developer can then peruse. In the same way, the abstract model can be sent to a validation module to check that desired properties (e.g., data liveness) hold.

Circe's interface is exposed to its users through a standard web browser. Beside providing convenient rendering facilities, the adoption of a web-based interface encourages clean separation between client and server operations, and permits easy integration of Circe's views with popular inter-personal communication methods (i.e, e-mail, discussion groups, publishing on web sites, etc.).

2.2 Parsing NL requirements

Parsing of natural language requirements is accomplished by the Cico domain-based parser [3]. Cico interprets NL sentences according to a set of rules classifying sentence fragments. Rules matching are evaluated according to a fuzzy metrics, so that in most cases even incorrectly-formed sentences can be parsed — at least in part — with a certain degree of confidence.

Both basic *terms* declared in a *glossary* (e.g., names of entities in the domain) and sentence fragments (describing complex entities) are classified according to their properties that are significant for the domain. Exploiting domain-specific knowledge already in the parsing stage makes the parser faster and more precise, reducing the scope for ambiguities. *Tags* describing domain-specific properties are assigned by users to basic terms. For example, an `/ELAB` tag means that the term so classified can perform computations, an `/EVT` tag describes the occurrence of an event that is significant in the domain, and a `/TSPAN` tag marks a term or expression indicating some length of time. Tags can also be inferred by the system by looking at the context in which terms occur in the requirements, a technique known as *speculative tagging*.

The output of the parsing stage is stored in the system's repository for subsequent retrieval and analysis.

2.3 Models and validation

The current version of Circe provides over 50 models of a software system. We use in this case study only a fraction of the available models; see [8] for a case study involving different models.

The models we use can be collected in three families:

- **Entity models** describe which entities are present in the problem domain, detail their properties (attributes) and the relations among them. Entity models do not express behaviour or function, but may model the nature of data on which behaviour depends and that are operated upon by functions. In particular, the structure of data describing real-world entities can be obtained directly from entity models. Certain relationships (e.g., *is-a* and *part-of*), whose semantics is known, are treated specially and taken into account when checking validation properties.

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

- **Behaviour** and **time models** describe the causal and temporal relationships between the events that can happen in the domain and the actions the system can perform. Behaviour models concentrate on describing, validating and measuring these aspects, abstracting from the nature of the events and actions that are introduced in the requirements.

Most models have formal internal and external (i.e., intra- and inter-model) consistency rules that can be checked by Circe. For example, data repositories in data flow models must be accessed to be useful, different parts of the system must communicate in some way, events must not fire conflicting actions, etc. We will introduce in more detail some model and associated validation properties in Section 4, where they are used to refine the LCS requirements.

2.4 An example

As an example, consider the following natural language requirement, taken from our translation of the LCS Problem Description document:[3]

> If a room becomes occupied before T1 (of the room) minutes since it had become unoccupied, the system sends to the light controller of the room the last selected scene of the room.

CIRCE's modelers query the parsed requirements repository looking for parse tree fragments that are significant for their respective models. In our case, a number of interesting facts about the system we are describing in the requirements can be extracted from the parse tree, shown graphically in Figure 2. Among the others:

1. A room is an entity that has a state, and can change its state at some point. The set of admissible states includes *occupied* and *unoccupied*[4] (from the becomes nodes).

2. A room is an entity with attributes. The set of attributes includes *T1* and *last scene* (from the attr nodes). CIRCE can check that these attributes are read from and written to — unless they are explicitly declared built-in.

3. Some relation exists between a room and a light controller; this relation is not given a name in this requirement (from the anonsel node). In order to interpret the expression "the light controller of the room", this relation must be navigable (from room to light controller) and unique (to avoid ambiguity).

4. A communication path exists between the system and light controllers (from the sends node). This path carries at least information of type *light scene*.[5] Moreover, the *light scenes* that the system outputs to the light controllers must be either built-in, or written to as described in some other requirement.

5. The system must be able to detect whether a room becomes occupied or unoccupied in order to fire the action part of the conditional statement. Moreover, it must include a clock component that can count at least 10 minutes (from the before and since nodes).

[3] Actual requirements input into CIRCE are in Italian, and are translated back to English here for readers' convenience, preserving as far as possible the original structure.

[4] As a validation opportunity, CIRCE can verify that entities with a state can enter and leave all their states in at least one case, warning the user when states that are never entered or never left are found. In our case, *room* is not a part of the system, its transitions are not under our control, and thus this kind of validation is skipped. Other validation checks that are performed in any case are: ensuring that entities with a state have at least two states, and — in presence of (optional) explicit declaration of states – that all the states declared are mentioned in requirements describing the behaviour of the given entity.

[5] Again, many opportunity for validation exists here. For example, CIRCE checks that the requirements do not describe communications among non-system components (that would be only noise for our purposes).

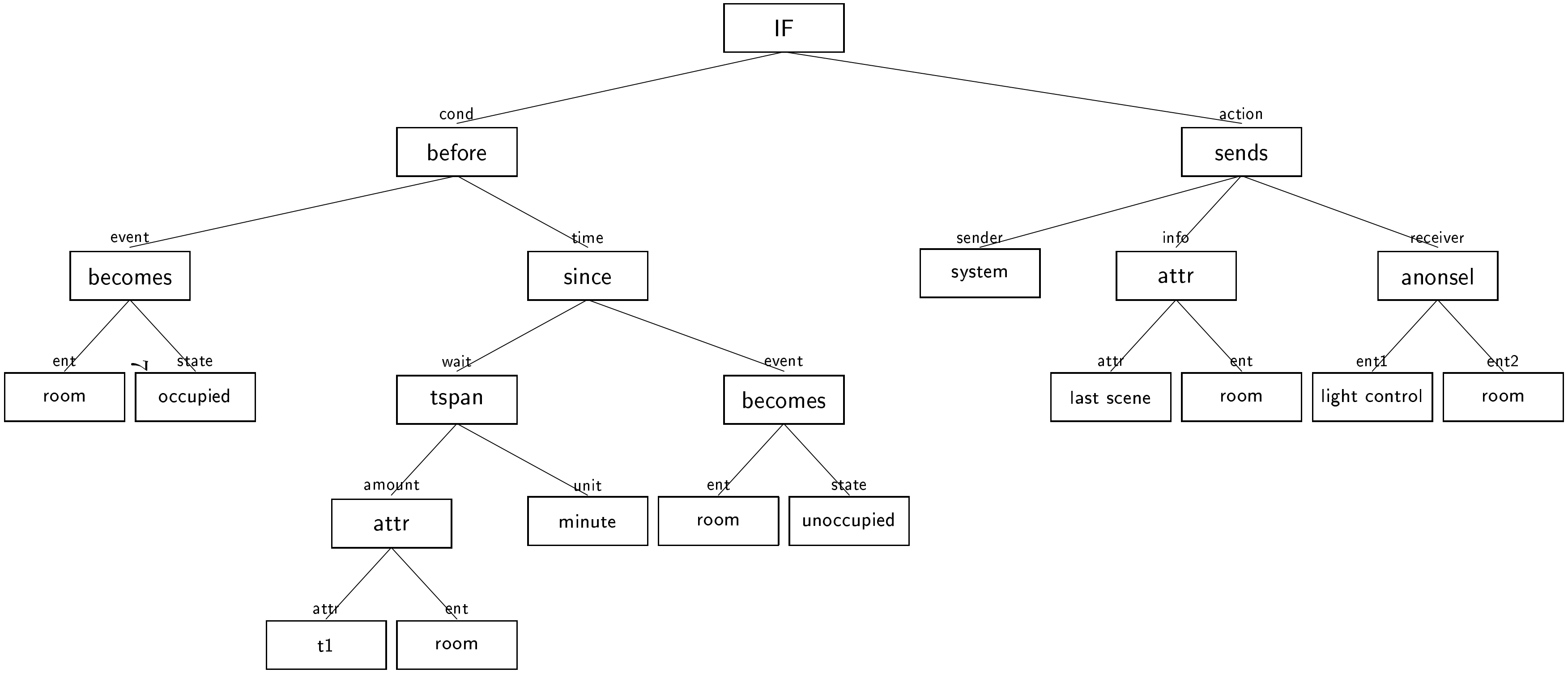

Figure 2: Parse tree for the requirement from page 6.

6. The set of possible behaviours of the system includes the complex one described by the whole requirement (from the IF node).

Other, less apparent deductions can be made. For example, the facts mentioned above also introduce some data, functional, and behavioural complexity in the system that can be measured and traced during the requirements writing process [4]. The requirement also introduce implicit design links among all the different details of different models described above. Changing the requirement in later stages can make some detail inconsistent or no longer significant: building models directly from the NL text of the requirements avoids the need to explicitly maintain design links among the different models.

3 Our approach

The setting of our case study was slightly atypical, due mainly to logistic consideration. In fact, most of the requirements elicitation process from the actual customer had already been conducted before our involvement.

The Problem Description Document (PDD) contained the *user requirements* ("informal needs") for the Light Control System (LCS). As can be expected in a similar case, user requirements were often vague and ambiguous (e.g., "no hazard for persons is allowed"), and in some case contradictory requirements were stated. Still, the PDD was already a much more refined document with respect to typical industrial cases, as it was prepared by requirements professionals (acting as a proxy for the real customer in subsequent phases). Moreover, an implementation of the system had already been built and accepted by the customer, and could be used as a final reference for a specification.

In this case study, our main goal was not to validate the PDD as it stood, but rather to incrementally translate it into a precise and consistent NL requirement document, using CIRCE to support the translation and validate the resulting document. The refined NL document could then be read (and understood) by the customer, and serve as a solid foundation for subsequent formal specification using any appropriate technique. In the terms we introduced in Figure 1, our approach amounts to conducting most of the requirements analysis and refinement in the Problems domain — that is, in terms comprehensible and thus more readily controllable by the customer — using automatic tools to map NL requirements into more formal representation in the Methods domain, to help the developer in analysing and validating them.

In our case, the customer was not readily accessible for cooperation on writing a more precise requirements set, so we used CIRCE in a single user setup as a modeling and validation tool. In this setup, editing cycles (see Figure 3) consists in editing a NL requirements document and submitting the document to CIRCE. The document is parsed by CICO, and CIRCE's modelers use the parsed requirements repository to build models of the system described by the requirements. These models, and the outcome of the validation that CIRCE

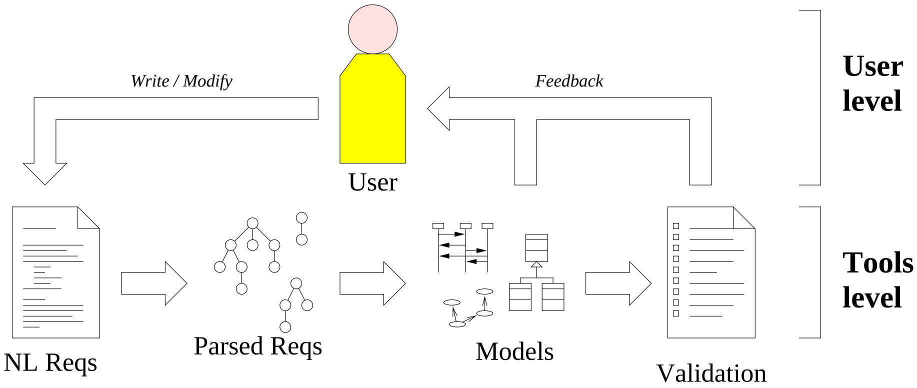

Figure 3: The write-validate cycle.

performs on them, are presented to the user and serve as feedback prompting further refinement of the original requirements.

Most of the steps in this cycle (NL parsing, model building, validation properties checking) happens at the tool level, and are entirely automatic. This is particularly important since automatic steps are repeatable, less error-prone,[6] and are performed efficiently. Very fast write/validate cycles — in the order of minutes — become thus possible, to the advantage of the human level steps (analyzing models and deciding which changes are needed in the requirements).

4 Translating and validating the PDD

The PDD was divided in four parts. The first one detailed the motivations for the project and introduced typographical conventions used in the rest of the document. The second part described the physical layout and the available infrastructures for the 4th floor of Building 32 of the University of Kaiserslautern campus, where the LCS was located. Part 3 listed informal needs for the LCS, separating functional needs and non-functional needs. Functional needs included user needs and facility manager needs. Non-functional needs included fault tolerance, safety and legal aspects, and user interface issues. A dictionary of terms of the application domain, in Part 4, concluded the document.

4.1 Initial glossary

The first step in our translation was to populate the glossary with terms from Part 4 of the PDD. Almost no tags were assigned in this phase, except for a few terms (e.g., "control system", "control panel", etc.) that were assigned the `/SYS`

[6] This is particularly true of clerical activities like checking a long list of properties. When similar activities are performed by humans, errors can be injected out of mere boredom or distraction.

tag to mark them as parts of the system we were asked to state requirements for.

Tags to many terms were assigned in later steps, either because requirements parsing needed specific tags, or via speculative tagging. For example, requirement U9.(iii) states that "The room control panel ... should contain ... a possibility to set T1", indicating that T1 is a piece of information that can be assigned a value, prompting us to assign the `/INF` tag (i.e., "this term designates a piece of information") to the term T1 in the glossary.

4.2 Domain description

The second step consisted in further describing the domain by translating — often almost literally — statements from the second part of the document in terms of known or newly-introduced relations. For example, the original statement in the PDD

> The fourth floor of Building 32 consists of three sections and shares two staircases, staircase east (SCE) and staircase west (SCW), with other floors of the building, as shown[7] in Figure 1. Each section is divided into some hallway sections (H) and rooms, each of which may be an office (O), a computer lab (CL), a hardware lab (HL), a peripheral room (P), or a meeting room (M). All rooms in a section are accessible via a connected hallway section. There are three hallway sections and 22 rooms to control.

was translated into

1. The fourth floor of Building 32 consists of three sections and two staircases.
2. Each section is divided into some hallway sections and a number of rooms.
3. Offices, computer labs, hardware labs, peripheral rooms and meeting rooms are rooms.
4. All rooms in a section are accessible via a hallway section.
5. The system controls three hallway sections and 22 rooms.

Part of the original description was omitted in our version. The schematic drawings illustrating the architecture of the floor (Figure 1 in the PDD), and the tables detailing geometric, physical and electrical characteristics of the analog sensors used by the systems were not translated into requirements. Most of the information contained in the omitted parts was indeed important for actual implementation (e.g., for writing hardware drivers for the sensors), but not relevant to the requirement phase. Abbreviations (e.g., CL for "computer lab") were simply introduced as synonyms in the glossary.

In performing this step, we introduced a few more terms in the glossary (e.g., the *is_accessible_via* relation) that were not included in the document dictionary.

[7]In the original Problem Description document.

Missing terms were immediately identified by CIRCE's document-related views, in particular through parsing failures and abstraction finding [9]. In a few cases when the correct tagging was not immediately apparent, speculative tagging was used to determine the correct set of tags.

The 23 statements resulting from this activity constituted a reasonably accurate description of the domain in which the LCS operate.

The models built on this description could be inspected in graphical form (see for example Figures 4 and 5) both to verify our understanding, and to check for possible incompleteness in the PDD. A few stylistic problems were already discovered in this phase: for example, a number of properties were predicated individually on each kind of rooms, while others were generically referred to the entity "room".

The *Subsumption* modeler identified such cases. The modeler examines the classification hierarchy built by other modelers on the basis of the *is-a* relationship among system entities, and identifies cases where an attribute or a relation common to all instances of a class can be lifted from the instances to the class. Moreover, the modeler also identifies opportunities for *approximate subsumption*, i.e. cases when a common attribute or relation is shared among a certain percentage of (but not by all) instances of a class.

The same modeler also identified an asymmetry between Peripheral rooms and all other kinds of rooms. In fact, the PDD described the rooms in five distinct sections, one for each kind of room. Offices were described in detail:

- **Offices.** Each office has one door to the hallway section and can have up to two doors leading to its adjacent rooms. Each door is equipped with a door closed contact.

  Each office is equipped with:

  - one motion detector [...];
  - two ceiling light groups [...];
  - [...]
  - two status lines [...].

Other rooms, however, were only described by difference from the offices,[8] i.e.,

- **Computer lab.** Same as office.
- **Hardware lab.** Same as office, but with more than one door leading to the hallway section.
- **Meeting room.** Same as office.

[8] CIRCE does not currently offer any language construct to express this type of "copy with modify" semantics, thus in the translation the similarity was expressed by collecting relevant rooms in a single statement, e.g. "Each office, computer lab, hardware lab and meeting rooms is equipped with ...". Notice that the "same as ... but ..." linguistic device is *not* expressing inheritance, for which CIRCE has appropriate constructs.

- **Peripheral room.** The peripheral rooms will not be controlled by the control system, and thus they are not described here.

In the approximate subsumption model, peripheral rooms were singled out for not participating in any relationship (and not having any attribute) shared by the other rooms. A better description would have been obtained by distinguishing *controlled rooms* (offices, labs and meeting rooms) and *other rooms* (peripheral rooms). The subsumption model would then suggest lifting all the relationships and attributes common to all *controlled rooms* (i.e., everything except for the number of doors leading from a room to the hallway) to the *common rooms* class, and some of these to the general *room* class.

It is interesting to notice that the simpler classification hierarchy for rooms established by the customer in the PDD led to serious interpretation problems in other parts of the document. For example, at some point a distinction was made between "offices" and "remaining rooms". A strict interpretation would have read "remaining rooms" as "all the rooms that are not offices", i.e. computer labs, hardware labs, meeting rooms, and peripheral rooms. It was not immediately clear from the context that indeed "remaining *controlled* rooms" was intended, and thus that the properties described did not apply to peripheral rooms. The more precise hierarchy suggested by CIRCE would have avoided this kind of ambiguity in the first place.

Other minor issues in the PDD (and in our translation of the PDD) were also identified by the *Selection* modeler. We collect under the label "selective expressions" all those linguistic devices that are used to select a specific instance from a set of similar entities. For example, the expression "the control panel of a room" selects a particular control panel (the one installed in the given room) from the set of all control panels in the universe of the LCS. The selection modeler can check the validity of selective expression by navigating the entity-relationship diagram built by other modelers, and verifying that a single navigable path exists among the referred entities.

For example, a possible ambiguity was discovered in the expression "the motion detector of a door", that admitted two different interpretations:

- the motion detector placed above a door, or
- the motion detector that detects motion in proximity of a door.

Common sense and some domain knowledge[9] would let us conclude rapidly that both interpretations select the same motion detector, but CIRCE's extreme ignorance of the ways of the world makes it a stricter analyst [5]. In fact, examining the rationale provided by the tool for its interpretation, we are reminded that each door has *two* motion detectors, one on each side, and thus the expression "the motion detector of a door", in itself, is seriously ambiguous. We need further inference (e.g., the fact that the system controls lights in hallways but not

[9] In this case, knowing the fact that motion detectors can only detect motion in proximity of the place where they are installed.

in staircases) or further context (e.g., the fact that a single hallway section is being considered) to interpret such a statement correctly.

In one case, the selection modeler identified a missing relation between "status line" and "hallway", whereas a statement

- The status line of a hallway shows the status of the ceiling light group of the hallway section.

was used. Indeed, in the translation process we had erroneously omitted part of a statement establishing an *is_equipped_with* relation between hallways and status lines. Similar occurrences show the importance of checking properties on the product of the requirement process as a means to remedy to errors (even clerical, as in our case) in the requirement writing process.

4.3 System description

Functional needs from the third part of the PDD were translated into requirements as for the domain description statements in the second part. However, while domain description statements were mapped almost exclusively to declarations of relationships and attributes, functional needs were mapped largely on functional- and behaviour-oriented requirements.

Language frames recognized by CIRCE do not allow the expression of patently incomplete requirements. To obtain correct parsing of the requirements, in several cases we had to complete the literal translation of the PDD with more explicit information. For example, the Facility Manager need

FM7 If a malfunction occurs, the facility manager has to be informed.

was translated as

FM7 If a malfunction occurs, the system sends a malfunction warning to the facility manager control panel.

Notice how the passive voice sentence "the facility manager has to be informed" was translated into its more detailed, active voice version above. This is an instance of the *completion* stereotypical operation on requirements [10]: information is added to fill the knowledge slots in a given template. In our case, we had to add three distinct facts to the original requirement:

1. the fact that the system is the initiator of the action (and thus responsible for verifying the condition),
2. the fact that interaction with the facility manager is through the facility manager control panel, and
3. the fact that the facility manager is "informed" through a warning message.

Another example of translation with completion comes from user need U5:

U5 The push buttons for the window- and wall-ceiling light group in a room should show the following behaviour:

(i) if the corresponding ceiling light group is completely on, it will be switched off;

(ii) otherwise it will be switched on completely.

that is translated as

U5 If the push button of a room is pushed,

(i) if the light group corresponding to the push button is completely on, the system sends the "switch off" command to the light controller of the room;

(ii) otherwise the system sends the "switch completely on" command to the light controller of the room.

Another stereotypical operation on requirements that was used throughout the translation was *linguistic refinement*. Refinement consists in substituting a high-level atomic term with a lower-level, structured expression. For example, in FM7 above, "a malfunction occurs" is defined as an atomic event (i.e., it is assigned an `/EVT` tag) in the glossary. Through refinement, the same event can also be described as

When a component is malfunctioning ...

where "malfunctioning" is a possible state (identified by a `/MODE` tag in the glossary) for every "component" of the system. Notice that in refining the original event we have introduced more information that is analyzable by CIRCE, in this case the fact that our system is made of components and that components can be malfunctioning. This results in more precise definition of our requirements. For example it is now clear that the expression "a malfunction occurs" only concerned malfunctions *of system components*, that are explicitly listed in the requirements, and not malfunctions of other entities in the domain.

Moreover, more complete validation is possible on refined requirements. In our example, validation conducted by the *BadAutomata* modeler on a Finite State Automata model of entities that can change their state would warn that "components" should have more than a single state, thus prompting us to write a full declaration stating that components can be either malfunctioning or working correctly. After this declaration, the same modeler would point out that it is not described anywhere in the requirements what happens when a malfunctioning component is repaired, and begins working correctly. In fact, this issue is never mentioned in the PDD, although a competent designer could figure a reasonable implementation (e.g., periodically polling malfunctioning components to check whether they have been repaired, or implementing a "component repaired" command for the facility manager).

Systematic treatment of requirements as shown above has the added benefit that hypothesis made during the translation (in particular, in refinement and completion) can be documented and checked with the customer or other interested parties.

Translation of the functional needs in the PDD resulted in 30 requirements of varying complexity. To these, translations of non-functional needs were added.

Some of the non-functional needs could not be expressed in terms of models known to CIRCE (e.g., NF8: "The control panels are easy and intuitive to use"), and were simply inserted in the requirements document (although not parsed) for reference. The remaining needs were translated as above, often in functional terms. For example,

NF2 If any outdoor light sensor does not work correctly, the default light scene for all rooms is that all ceiling light groups are on.

was translated as

NF2 If the outdoor light sensor of a room is malfunctioning, the system resets the default light scene of the room to the emergency light scene.

Notice how in this translation we were induced to complete the requirement by transforming the impersonal "the default light scene ... is that all ceiling light groups are on" to an explicit action performed by the system. Notice also how we introduced an explicit "emergency light scene" corresponding to (but conveying more information than) setting all light groups to on.

Our version of requirement NF2 also introduces some correction originated from verification of our understanding with the customer. In fact, the original form of the requirement specifies that *all* rooms should have their lights on if *any* outdoor light sensor fails. This is excessive, and conflicts with other non-functional requirements asking to minimize electrical power consumption. However, it is formally consistent, and thus no amount of formal validation could flag it as an error.

The customer's real intention was that the system would turn on the lights on all those rooms for which the calculation of available daylight depended on the malfunctioning outdoor light sensor (there were six sensors, corresponding to different orientations of the rooms' windows, as shown in Figure 1 of the PDD). Hence, we used the selective expression "of a room" both for the failing sensor and for the light scenes. After this correction, however, CIRCE's validation modelers signaled the missing relationship between outdoor light sensors and rooms (a relationship that was implicit in that figure, but not stated anywhere in the requirements). Explicitly declaring the "same orientation" relation solved the problem, making requirements more precise once more.

4.4 Dictionary and definitions

Finally, the fourth part of the PDD, the Dictionary, was translated, including terms definitions. Of the 40 terms comprising the dictionary, 37 were already in CIRCE's glossary. The remaining three terms were either not needed in the requirements, e.g., "architecture: structure of a building, floor or room", or falsely atomic, e.g., “hallway sections ceiling light group”. In place of the latter term, we had used the selective expression “ceiling light group of a hallway section”. We had also used the corresponding “ceiling light group of a room” where needed, while the PDD did not contain the symmetric term “rooms ceiling light group”.

A few definition called for explicit requirements, mainly to establish appropriate classification hierarchies. For example, from the definition:

> Motion detector: *sensor detecting motion of a person or animal in its range.*[10]

we extracted the domain description statement:

> Motion detectors are sensors.

(we could also extract the fact that motion detectors have an attribute “range”, but this attribute was never needed in the requirements document). Explicitly introducing sensors (and actuators for symmetry) also simplified our listing of the “components” of the system in the requirement:

> Sensors, actuators, and control panels are components.[11]

4.5 Effort needed

At the end of the translation process, our requirements document consisted of 62 statements, of which 28 were domain description statements, and 34 expressed functional and non-functional requirements.

The entire translation step required three days of work, including the time needed to familiarize with the domain. No source of information outside the PDD was taken into account, and no customer interaction was available during that time. Since the translation proceeded linearly, we had to revise several times our understanding of a certain user need in light of information that was supplied later in the PDD. We believe that effort could have been substantially reduced had the customer been available for comments during the translation process.

5 Validation findings and customer feedback

As discussed above, the process of translating statements from the PDD into requirements accepted by Circe uncovered a certain number of defects in the original document. These defects were in part identified by the validation modelers in Circe's library, and in part by the effort of rewriting statements to fit Circe's linguistic templates.

[10] This definition is obviously approximate: there is no way a motion detector could distinguish a moving person or animal from anything else, say a moving robot. This however is typical of customers' problem descriptions, and mostly irrelevant for our purposes.

[11] Notice that we did not include the core control system as a component, since malfunctions in the core control system cannot be handled by the control system itself.

Every assumption made in resolving the issues in the PDD was recorded for subsequent verification with the customer. In some case, later assumptions dominated previous ones, and the former were deleted. Only problems that hindered the faithful translation of the document were considered at this stage; problems with the system described by the requirements were left for a later step. Overall, slightly less than 50 assumptions and open questions were left at the end of the translation phase. Of these, 16 were selected as high-priority issues, i.e., issues that could not be resolved without further information from the user, or cases in which one of several possible solutions was opted for in taking some assumption.

The 16 most relevant questions and assumptions were then sent to the customer for answer and verification. The customer replied to only 12 questions, confirming the assumptions we made in eight of them. In the remaining four cases, information missing from the PDD was provided (e.g., we discovered that each door had a small window above it through which light could fall into the hallways).

The four questions to which the customer did not reply were mainly concerned with the structure of the document (rearranging sections, merging requirements) and with definitions in the dictionary. Although no comment was obtained on these four questions, a subsequent version of the same document was arranged as suggested by three of them, while the fourth question was no longer applicable (a term whose definition was questioned had been removed from the dictionary).

Customer interaction convinced us that most of the problems we had discovered in the process of translating user needs into requirements, and then through the validation modelers we had available, were real defects in the problem description document. Preliminary results seem to indicate that a significant portion of those problems were also independently identified by other researchers working on the same case study. Unfortunately, precise comparative data were not available to conduct a formal assessment of the effectiveness of our technique with respect to other approaches.

6 Subsequent activities

Our goal in this case study was to test the effectiveness of CIRCE in analyzing and refining customers' documents describing a problem and its domain. It was not our intent to do a complete requirement analysis on the Light Control System. However, in this section we point out some of the work that was left for later stages, by using CIRCE's views to establish a roadmap of activities.

Figure 4, for example, shows a simplified entity-relationship view of the system described by the (translated) requirement document. In that figure, labels for relations and attributes have been hidden to avoid excessive clutter, and some basic relationship (i.e., is-a and part-of) have been filtered out for the same reason.

The automatic graph layout algorithm used by CIRCE's graphical views has

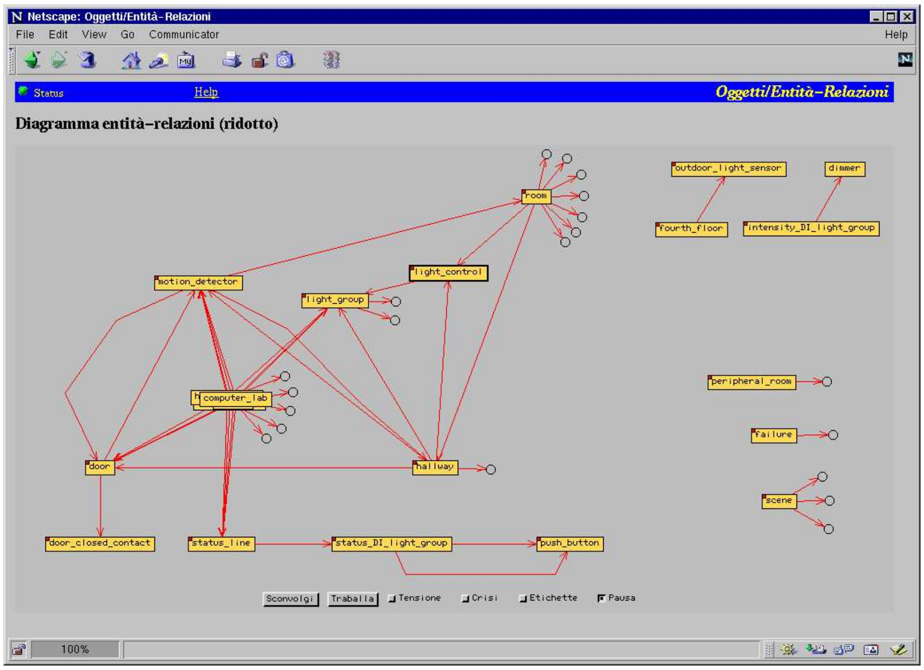

Figure 4: E-R view for the translated PDD.

placed a number of entity nodes in the same position (the stack of nodes that has "computer lab" on top). It can be easily observed that all those nodes, one for each kind of room, share the same relationships with other nodes in the diagram. Indeed, this observation corresponds to the approximate subsumption modeler suggestion that such relationships could be moved to the parent class "room", if not for the "peripheral room" that can be observed, isolated, on the right hand side of the picture.

One of the first actions that should be taken in further refining the requirements document would then be to reorganize the class hierarchy for rooms, either by introducing new intermediate classes as suggested in Section 4.2, or by dropping peripheral rooms from the description of the domain. This refinement alone would greatly simplify the entity model.

The graph in Figure 4 is broken in six components, probably too many: an indication that the domain description is not accurate enough. When is-a and part-of relations are shown, however, the graph becomes more strictly connected. Still, the outdoor light sensor entity does not participate in any relationship beyond the simple "is part of the equipment for" with the "fourth floor" entity. In fact, a relationship expressing the fact that a room has the same exposure as a given outdoor light sensor should be added, as mentioned above.

While domain description could be marginally refined beyond what we have

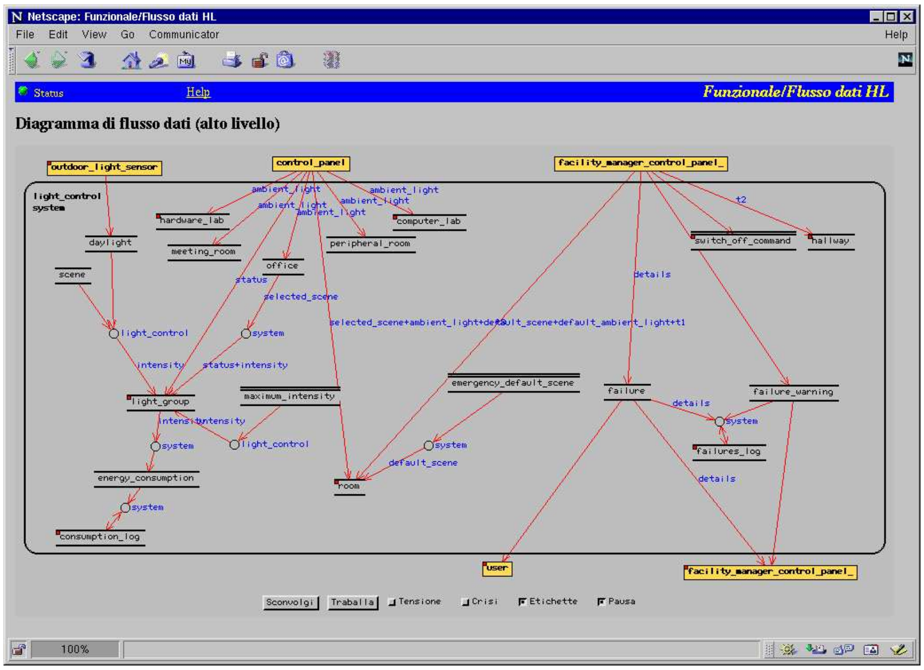

Figure 5: High level data flow view for the translated PDD.

already suggested, the inner workings of the system are only sketched up to this point, and require significant work to be brought at least to internal consistency. This is not surprising, given that our requirements document only contains direct translations of user's needs, and is not detailed enough yet to guide the design phase.

Figure 5 shows the high level data flow diagram for our system. As can be observed, most flows in the system are incomplete, and the only output described explicitly are warnings sent to the user of a room and to the facility manager when some component of the system fails to operate correctly. Most repository are never brought outside the system. In particular, the PDD did not describe any way to access the energy consumption log (on the bottom left corner of the system box in Figure 5) and the failures log (on the right, above the facility manager control panel). Moreover, considerable asymmetry can be observed between offices and other rooms. Settings for the light groups are in certain places described as a pair (status,intensity) computed according to the selected scene of an office, and in other cases as a simple intensity value computed from a generic "scene" (taking into account daylight intensity measured from outdoor light sensors).

While some of the unpleasant features in the data flow model may have been introduced by inaccurate translation, most of them are a direct result of ambiguities, inconsistencies, and underspecifications in the PDD.

All these potential problems, and others, should be resolved in order to obtain a requirements document that is precise and detailed enough for the design phase. The possibility to access interactive and dynamically updated diagrammatic representations of models of the system described in the requirements document is in our opinion an invaluable tool to rapidly identify problems[12] and evaluate possible solutions. At the same time, the availability of a textual, natural language description of the system simplifies interaction with possibly non-technical customers and other interested parties, thus simplifying verification.

7 Lessons learned

In performing this case study, we have tested CIRCE's ability to support the activity of writing requirements for a new system even in its initial stages. The mostly literal translation of statements expressing user's needs about the LCS was sufficient to CIRCE to build an initial model of the new system and of its environment. Validation modelers were used both to identify a number of problems in the original customer's document, and to plan a roadmap for further refinement and completion of a more precise requirements document. Interaction with the customer confirmed that the vast majority of the problems identified in the PDD were indeed real problems needing clarification or further explanation.

The case study also served to indicate opportunities for the extension of CIRCE's library of modelers towards the handling of non-functional requirements. The extension does not impact linguistic aspects (in fact, rules to parse non functional requirements could be easily added to CICO's rule set), but rather reflects some immaturity in current practices when it comes to building and reasoning on models of non-functional requirements. Extending CIRCE's capabilities in this direction requires defining new models for certain families of non-functional requirements, together with their associate representations, validation properties and possibly metrics. In this spirit, [2] describes a possible model for structural requirements, i.e., non-functional requirements that have an impact on the architecture of a software system.

While a complete assessment has not been conducted, comparison with other reported experiences [6] seems to indicate that the support offered by CIRCE to the requirement writing activity scores comparably with other techniques in terms of accuracy, while requiring less expertise and effort on the part of the requirement writer. Moreover, since the final product is a natural language requirements document, user validation can be performed without the need for translating back formal requirements into a language comprehensible to the user (a translation that could inject insidious errors in the process). We believe that

[12] All the issues we have discussed, with the exception of asymmetries between offices and other rooms, are also identified by validation modelers. This makes it possible to only examine diagrams when problems are identified during the requirements engineer's work, and automatize partial formal validation of a system.

this fact can substantially improve interaction between the requirements writer and the customer, and that improved interaction leads to requirements of better overall quality.

8 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.

One of such systems is the "Requirements Apprentice", by Reubenstein and Waters [14, 15]. 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 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. The system does not explicitly discriminate between syntactic and semantic aspects of the requirements, i.e., it lacks distinct notions of model and view; this is reflected by the small (and fixed) number of representations available. Operating on the repository, a hybrid reasoning system called Cake [7] can detect contradiction and incompleteness, disambiguate ambiguous sentences, and provide natural language output, thus producing actual requirements documents. The function of Cake is similar to that of validation modelers in Circe, but seems to provide less functionality.

In a more recent work, Maiden presents the AIR Toolkit [11]. 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 parser with a more complex structured editing interface, and the Requirement Critic as a validation modeler coupled with an interactive representation.

Johnson et al., in [10], 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, this knowledge base maps onto our parsed requirements repository). ARIES can then present this knowledge to the user either in textual form (by using the GIST explainer [16]), 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. 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 [12], a system designed to test the limits of distributed requirements engineering. In contrast with Circe, where a centralized repository is accessible by every modeler, in The Viewer each modeler (embodying a viewpoint [13]) 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 projector; 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.

While many more systems have been proposed in the literature than we can cite here, as far as we know no proposal exists that couples free-form natural language parsing with the modeling and validation capabilities that Circe provides through its domain-aware modelers.

References

[1] V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proceedings of the Twelfth IEEE Conference on Automated Software Engineering*, pages 36–45. IEEE Press, Nov. 1997.

[2] V. Ambriola and V. Gervasi. Representing structural requirements in software architecture. In R. Nigel Horspool, editor, *Proc. of the IFIP WG2.4 Conference on Systems Implementation 2000: Languages, Methods and Tools*, pages 114–127. Chapman & Hall, Feb. 1998.

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

[4] V. Ambriola and V. Gervasi. Process metrics for requirements analysis. In *Proceedings of the Seventh European Workshop on Software Process Technology*, Lecture Notes in Computer Science. Springer-Verlag, Feb. 2000. (to be published).

[5] D. M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, Feb. 1995.

[6] E. Börger, B. Hörger, D. Parnas, and D. Rombach, editors. *Requirements Capture, Documentation, and Validation*, 1999. Dagstuhl Seminar Report 242.

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

[8] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements. In *Proc. of the 4th International Conference on Requirements Engineering*, June 2000.

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] P. Zave. Call for papers and associated classification scheme. In *IEEE International Symposium on Requirements Engineering*, 1995.