On the parallel refinement of NL requirements and UML diagrams

Vincenzo Ambriola Vincenzo Gervasi
Dipartimento di Informatica, Università di Pisa

1 Introduction

In the last few years, the UML has gained widespread acceptance as a modeling tool, becoming the *de facto* standard language for communicating domain models, system models, and software architectures. The focus placed on the expressiveness of the graphical notation is probably among the main reasons of this success. In our experience, the general meaning — although not the details — of most UML diagrams can be grasped by novices with very little specific training. Still, the support we currently have for writing UML diagrams, and especially for evolving them in a structured way, has not reached the point for novices to write in UML with relative ease.

In this paper we report on our experiences with the automatic synthesis of UML diagrams from natural language (NL) requirements in our requirements engineering environment Circe [2, 4, 5], with particular attention to the tool-supported parallel refinement of NL requirements and of the associated UML diagrams.

Our handling of NL requirements is outlined in Section 2, including some examples. Section 3 briefly describes the strategies we use to synthesize a number of UML diagrams from the information collected from the requirements, and shows how refinements applied through linguistic means to the text of the requirements are mapped to analogous refinements in the UML diagrams. The last section provides some conclusions.

2 From NL to UML

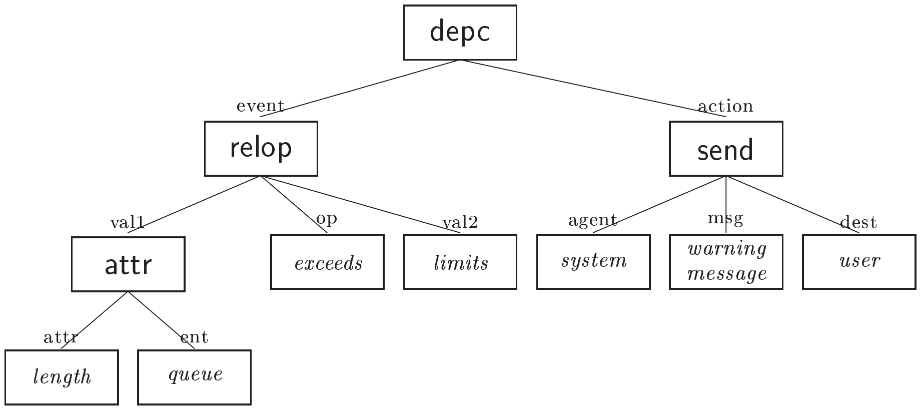
To extract information from natural language requirements, we use the domain-based parser Cico, a shallow parser implementing domain-based parsing [3] that is part of the Circe environment. Cico can also be described as an engine for the application of a fuzzy rewriting system to a text, using backtracking and heuristic optimization strategies to determine an optimal parse tree for a statement. For the purpose of this paper, it suffices to say that the parsing process transforms natural language requirements into abstract *parse trees* whose nodes represent the (domain-specific) semantic links between the constituent terms (see Figure 1).

From a forest of such parse trees, that represents the information extracted from the requirements, other components of Circe build a number of models of the system described by the requirements, e.g. data flow, entity-relationship, finite state automata, event-condition-action models. These models form the basis for the translation of requirements into UML diagrams.

The parser Cico and the modeling components of Circe interpret the requirements according to a precise view. In this view, a software system is seen as a set of communicating *agents*. Each agent can send, receive, and perform computations of various sorts on typed packets of *information*; an agent can also optionally have a current *mode* of operation (or *state*) from a given set of modes. Agents' behavior can be specified in terms of observable *events* and *conditions*, and of *actions* to be performed. Events include causal and temporal dependencies.

Any entity in the domain can have *attributes*.[1]. Relationships among entities can be declared and used to select specific instances from a generic class, e.g. "If a meeting is canceled, the system

[1]This includes both real-world entities, and software-only entities, e.g. internal representations that the system maintains. The value of such attributes is considered to be an information packet.



"Whenever the queue length exceeds the limits, the system shall send a warning message to the user."

Figure 1: A sample requirement and its domain-based parse tree, as produced by Cico. Leaf nodes represent basic terms from the requirement text; internal nodes represent domain-level relationships among terms (attr=attribute, relop=relational operator, send=message passing, depc=conditional dependency).

shall delete *all the room reservations for the meeting* from its database" (implying that there is a "reserved_for" relation between rooms and meetings).

Once all the different bits of knowledge about the system to be specified and its environment are collected from the requirements and consolidated in uniform models by Circe, the models are rendered to UML diagrams through systematic (and automatic) translation. Naturally, the UML diagrams thus obtained will be no more precise than the requirements from which they were obtained. In particular, in several contexts in which the UML does not specify a particular language to denote expressions (e.g., in cardinalities, in constraints, and in type definitions for some data item), the wording used in the text of the requirements is imported into the diagrams. The general principles guiding the translation are outlined below:

- **Objects and classes.** The whole system and its domain are modeled as a set of objects, with commonalities collected in classes. We distinguish several types of objects: *actors* represent external entities that interact with the system to be designed; *passive objects* represents software entities that describe real-world or software-only things; *active objects* add to this their own flow of control and the ability to initiate actions.

- **Relationships and attributes.** Objects corresponding to entities cited in the requirements are then enriched by adding to them the attributes and relationships inferred from the requirements. When not explicitly stated, cardinalities are heuristically inferred from linguistic features (e.g., the use of plural forms, of certain articles, of selective expressions). Also, directionality is determined by following the logical path of specifiers (e.g., "the room reservations of the meeting" implies that the reserved_for relation must be navigable from meeting to room reservation).

- **Data flows and computations.** The passage of information among entities in the requirements, together with any calculation described, is used to build an internal data flow model, that in turn is mapped into a set of operations in UML class diagrams (and related calls in sequence diagrams) that implement the required communications. In most cases, this entails adding an operation to the receiving object of a communication, or to the object that is the argument of some computation.

- **Dependencies.** Causal, temporal, and data dependencies extracted or inferred from the requirements are used to create an events-conditions-actions (ECA) model of the system de-

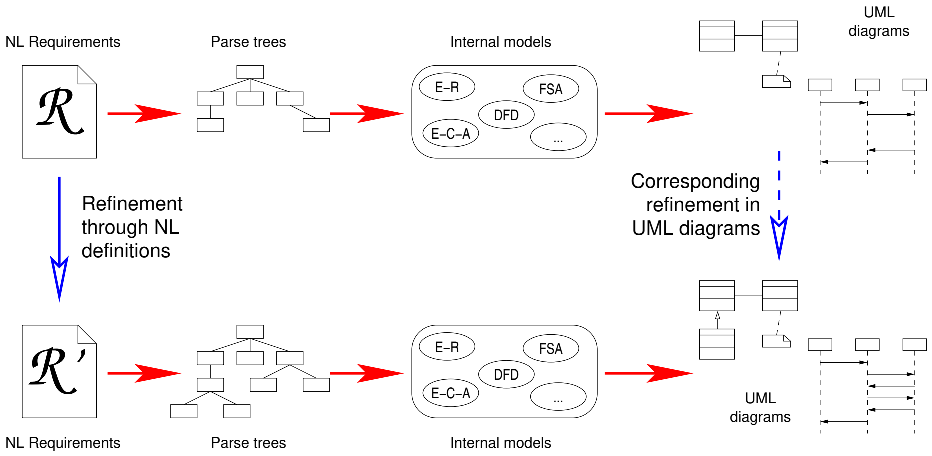


Figure 2: Overview of the transformation and refinement steps.

scribed in the requirements. From the ECA model, interaction and state diagrams are derived, describing most of the inter-object and intra-object behavior of the system.

While we do not have space here for the details, the general flavor of the translation can be observed in the example in Figure 3.

3 Parallel refinement

The UML *as a language* does not provide explicit support for refinement during the evolution of a software design. An approximation can be obtained by using stereotypes generously throughout the design, and then refining the definition of the stereotypes. However, this approach requires that the correct set of stereotypes to use is determined in advance; thus it is seldom applied in practice.

In our approach, refinement is applied at the requirements level, rather than at the design level, and correspondingly refined UML diagrams are obtained through automatic transformations by our tools, as shown in Figure 2.

Refinement at the requirement level is expressed through *definitions*. A definition explicitly states how a certain linguistic expression is to be interpreted by the reader, and is thus distinct from a *designation*, that simply assigns a name to a given basic concept in the domain [9]. As a particular case, the interpretation instructions given in a definition can explain how a certain function or feature of the software is to be implemented, and can thus express refinement. It is worthwhile to remark that in our case (i) definitions can be given *a posteriori*, making it unnecessary to plan beforehand the entire future evolution of the specification, and that (ii) they can be applied to any arbitrary piece of text, and thus are not restricted to a certain class of constructs.

In CIRCE, definitions are expressed as

$$model \quad \mapsto \quad substitution$$

with the intuitive meaning "whenever a fragment of text matches the *model*, replace it with the *substitution*". The *model* may contain constant parts (denoted by uppercase terms) and variable parts (denoted by lowercase terms). Variable parts may be supplemented with a set of tags (not shown in the examples in Figure 3) to restrict matching to only those terms whose semantic classes include the given tags. For example, `z/HUMAN` would match "the user" but not "the terminal" since only the former is designated as a human being.

In the example in Figure 3, taken from the requirements document for a GSM messaging system, we describe at first communications between the system and its user, as shown in Figure 3(a). Later, we *define* these communications in more concrete terms, introducing the terminal (i.e., a cellular

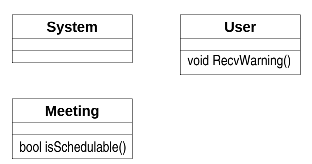
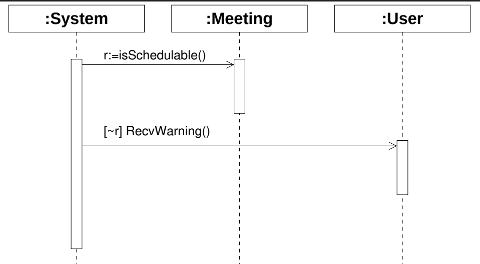
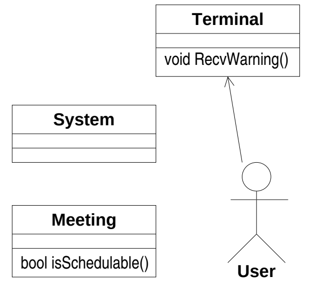
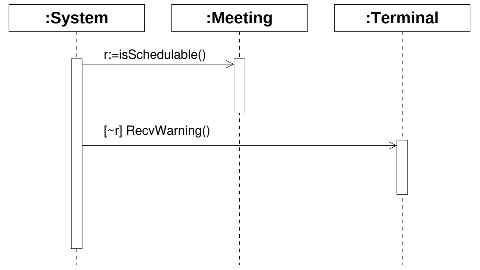
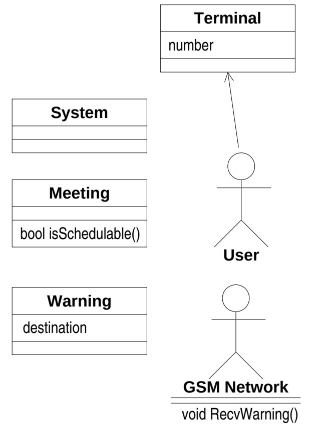
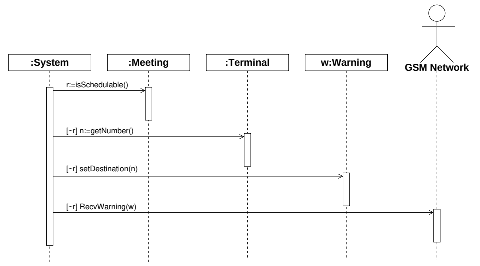
Requirement/Definition	Class diagram	Sequence diagram
(a) *The original requirement:* If the meeting is not schedulable, the system shall send a warning message to the user.		
(b) *We refine the requirement by writing the following definition:* x `SENDS` y `TO` z $\mapsto$ x `SENDS` y `TO TERMINAL OF` z *so that the previous requirement is now interpreted as if it were* If the meeting is not schedulable, the system shall send a warning message to the terminal of the user.		
(c) *Later, we refine even further by writing the definition* `SENDS` y `TO` w $\mapsto$ `SETS DESTINATION OF` y `TO NUMBER OF` w `AND SENDS` y `TO GSM_NETWORK` *yielding the transformed requirement* If the meeting is not schedulable, the system shall set the destination of a warning message to the number of the terminal of the user and shall send the warning message to the GSM network.		

Figure 3: Parallel refinement in NL and UML through NL definitions.

phone) of each user. Finally, as shown in Figure 3(c), we add a new definition, taking into play the GSM network and the need for explicit addressing of each message sent through the network.

As can be seen, while the requirement presents the system in logical terms of messages and users — to the advantage of understandability and modifiability —, the corresponding UML diagrams obtained by our tools through the definitions are detailed enough for analysis and implementation.

4 Conclusions

In this paper we have presented a combination of transformation techniques (from NL requirements to UML diagrams) and of structured refinement on requirements (through NL definitions) that can be used to automatically obtain refinement on UML models. Such refinements would be extremely difficult to obtain through UML stereotypes, especially in the common case in which a particular direction for refinement has not been planned in advance.

Similar results can be obtained by using Graph Grammars or other graph rewriting formalisms, in which the production rules play a role analogous to that of our definitions ([6] and [8], among others). However, we believe that the indirect refinement that we propose offers some advantage over direct rewriting, including (i) the guarantee that requirements and UML diagrams will stay consistent with each other, since the latter are automatically derived from the former, and (ii) the fact that NL definitions can be easily validated by the customer, since definitions are expressed in domain terms. On the contrary, in direct graph rewriting the customer does not get an opportunity to validate the refinement, since rewriting rules are expressed in software modeling terms.

The difficulties in direct rewriting outlined above have been recognized in the literature, and have led to a number of studies attempting to identify what constitutes a valid refinement and how to apply transformations in this context (e.g., [1, 7, 10, 11]). In our proposal, consistency is obtained as a by-product of the generative approach, while validity can be assured with the help of the customer.[2]

In our experience, both the use of NL requirements and definitions, and the availability of automatic tools for transforming NL requirements into UML diagrams, have greatly improved the ability of non-specialists to express, reason with and refine system descriptions in UML. While specialists could have performed the same refinements by directly editing their UML models, the use of definitions implicitly produces a complete trace in which every single decision has been clearly recorded and — more importantly — can be changed at any future stage, obtaining a new, consistent UML model.

References

[1] M. Abi-Antoun and N. Medvidovic. Enabling the refinement of a software architecture into a design. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 17–31. Springer, 1999.

[2] V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proc. of the 12th International Conference on Automated Software Engineering*, pages 36–45, Los Alamitos, November 1997. IEEE Computer Society Press.

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

[5] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements. In *Proc. of the 4th International Conference on Requirements Engineering*, pages 140–148, June 2000.

[6] M. Gogolla. Graph transformations on the UML metamodel. In J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, and J. B. Wells, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[7] M. Gogolla and M. Richters. Transformation rules for UML class diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 92–106. Springer, 1999.

[8] W. M. Ho, J.-M. Jezequel, A. L. Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. Technical-report, Inria, Institut National de Recherche en Informatique et en Automatique.

[9] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices.* ACM Press, 1995.

[10] L. Li. On managing classes for evolving software. In *Proceedings: Seventh International Workshop on Program Comprehension*, pages 144–150. IEEE Computer Society Press, 1999.

[11] T. Mens, C. Lucas, and P. Steyart. Supporting disciplined reuse and evolution of UML models. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 378–392. Springer, 1999.

[2]In addition to this user-level validation, the Circe environment provides a wide array of intra- and inter-model consistency checks, performed by our tools on the intermediate models [4] — *before* UML generation takes place.