Lightweight Validation of Natural Language Requirements: a case study

Vincenzo Gervasi
Dipartimento di Informatica
Università di Pisa
I-56125 Pisa, Italy
Email: gervasi@di.unipi.it

Bashar Nuseibeh
Department of Computing
Imperial College
London SW7 2BZ, UK
Email: ban@doc.ic.ac.uk

Abstract

In this paper, we report on our experiences of using lightweight formal methods for the partial validation of natural language (NL) requirements documents. We describe a case study based on part of NASA's specification of the Node Control Software of the International Space Station, and apply to it our method of checking properties on models obtained by shallow parsing of natural language requirements. These experiences support our position that it is feasible and useful to perform automated analysis of requirements expressed in natural language. Indeed we identified a number of errors in our case study that were also independently discovered and corrected by NASA's IV&V Facility in a subsequent version of the same document. The paper describes the techniques we used, the errors we found, and reflects on the lessons learned.

Keywords: *Natural language requirements, lightweight formal methods, requirements validation.*

1. Introduction: lightweight formal methods and requirements validation

The use of lightweight formal methods has recently received increasing attention in the software development literature [Feather 1998; Jackson & Wing 1996]. In the context of requirements engineering (RE), we use the term "lightweight formal methods" to characterise those methods whose adoption cost is a small fraction of that of the overall RE process, including training, application and computational costs. Lightweight formal methods often perform partial analysis on partial specifications only [Easterbrook *et al.* 1998]. They do not require a commitment to translate an entire (informal) requirements document into a formal one, nor to maintain formal and informal versions of specifications in parallel [Kemmerer 1990]. Moreover, as requirements specifications evolve during the early stages of the RE process, lightweight formal methods provide an opportunity for gradually validating requirements, paving the way for later introduction of more exhaustive and rigorous analysis if needed.

A number of experiences have been reported on the use of lightweight formal methods. These range from their application to the very early stages of a development process (e.g., [Goldin & Berry 1997] who use lexical analysis to find abstractions in unstructured and uninterpreted text), to design support systems [Hesketh *et al.* 1998], and to reengineering applications on existing code [Murphy *et al.* 1995]. Others have studied the application of NL understanding techniques to the automatic extraction of models from NL requirements [Rolland 1992; Macias & Pulman 1993]. The application of lightweight methods to the analysis and validation of NL requirements is particularly appealing, since industrial practice shows that NL requirements are easier to evolve, maintain and discuss with (possibly non-technical) customers. However, it is often very difficult to prove properties such as correctness, consistency and minimality about NL requirements. This paper describes a real case study demonstrating the practical application of lightweight methods to analyse such requirements.

The paper is structured as follows. We begin by providing some background to our case study, followed by a short presentation of our general framework for lightweight validation of natural language requirements. Then we describe the application of our framework to the case study, discuss our findings, and reflect on the lessons we learned. A short survey of related work and a discussion of future work conclude the paper.

2. The case study

We studied a fragment of a NASA Software Requirements Specification (SRS) for the Node Control Software (NCS) on the International Space Station [NASA 1997]. 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 subject to many inspections and revisions), and because parts of it had already been analyzed using different techniques, in related studies [Easterbrook *et al.* 1998*;* Russo *et al. 1998;* Russo *et al.* 1999].

The document, 250 pages long, is written mainly in narrative English, with several tables and the occasional schematic diagram interspersed in the text. The 3-page fragment we chose to analyze described one of the basic components of the Environmental Control function — Cabin Pressure Monitoring. The NCS continuously monitors the cabin pressure, and issues alarms if the measured pressure exceeds operating limits. This function can be disabled and enabled as part of Fault Detection, Isolation and Recovery (FDIR) procedures.

The document is structured by NCS functions (e.g., Telemetry Control, Environmental Control, Time Management, etc.). Each function is described in terms of individual constituent components (e.g., Environmental Control includes pressure monitoring, air fan control, fire & smoke detection, etc.). Each of these components, in turn, is first introduced in general, narrative terms, and then detailed by describing its inputs, outputs and expected behaviour.

3. Approach: validating natural language requirements

Our approach to automatic partial validation of NL requirements is structured in a *setup phase* and a *production phase*. The setup phase includes the following activities:

1. *Defining a style, a structure and a language for the requirements document.* This step can be meant either normatively, i.e. as the production of a prescriptive style manual for the requirements document (and in this case a syntax-guided editor can be used to support requirements writing), or descriptively, i.e. as an adaptation of the capabilities of a *parsing* tool to an already existing document written in a defined style (as in the case of the experience we report here).
2. *Selecting desirable properties to check.* Which properties of a certain document or system described in a document are "interesting" depends on the particular context of the analysis. As is common with lightweight formal methods, partial validation is usually acceptable at this stage.
3. *Defining one or more models on which the properties selected in the previous step can be checked*. Properties are always relative to models, i.e. abstractions of the document or of the system described in it, which collect in an analyzable structure the information needed to check the property. For example, a connection property among system components can be checked on a model describing all the communications among system components.

Once the setup phase has been completed, the production phase can be iterated at any stage of development of the requirements — without incurring any significant additional cost, as we will show later. The production phase of our approach includes:

4. *Pre-processing the requirements document*, to handle format, structure and typographical details and to translate the requirements document to a canonical form amenable to later processing.
5. *Parsing the NL text of the requirements*, leading to an analyzable representation of the semantic content of the text. Again, parsing can be (and usually is) partial, to help in reducing the cost of the validation, as long as that does not interfere with the collection of the information needed to perform the validation.
6. *Building the models* defined in step 3 above, using the information collected during the parsing process. It is possible to build models of the requirements document (for example, distribution of topics among sections of the document) and of the system described by the requirements (for example, a model of the communication paths in a distributed system).
7. *Checking that the models satisfy chosen properties.* As in the previous step, it is possible to check properties of the document (in our previous example, consistency of topics inside a single section) and of the system (for example, the existence of disjoint components in the communication paths model).
8. *Evaluating findings and revising the requirements specification accordingly.* It is particularly important that the validation checks provide as much detail as possible about the point and the reason of a failure (i.e., on the circumstances in which a validation property was violated). Similarly to the counterexamples provided by some full formal methods, this information helps the requirements engineer to identify and fix errors that cause violations.

This process offers a number of advantages in an industrial setting. Steps 1-3 are reusable across projects, as each organization tends to adopt defined internal standards for document style (step 1) and quality control (steps 2 and 3). Moving these standards into a tool is an effective way to accumulate the organizational knowledge and expertise in a safe and structured way, and to have it applied in a deterministic and reproducible manner during the production phase. Also, steps 4-7 are entirely automatic, leaving step 8 only for the requirements engineer to consider at each iteration.

4. Experience: application of approach to the case study

In this section we describe the details of our experience with the case study, according to the structure of the process outlined above. Since — due to logistic considerations — we had to work alongside NASA's standard verification and validation process, and not inside it, we only ran a single iteration of our production phase. Also, we analyzed three major revisions of the requirements document in "batch mode". This is not the best possible setting for lightweight validation, which is actually better suited for continuous application *during* requirements evolution between minor revisions. However, this unfavorable setting offered the opportunity to compare our findings with those of a traditional V&V process as performed by NASA (mostly inspection); the results of this comparison will be described in the next section.

4.1. Instantiating the approach

1. *Defining a style, structure and language.* The NCS specification exhibited a consistent style and structure (conforming to DOD-STD-2167A), and was of overall good *structural quality* [Fabbrini *et al.* 1998]. The language used in the detailed descriptions of each function was concerned mainly with (fairly complex) temporal ordering of input and output events, but also included user interface and other technical issues[1] that influenced the kind of language used. On the other hand, the narrative text was much more elaborate from a linguistic point of view, but since it was intended merely as an explanation of the technical text in the engineering part (that served as the definitive reference), it added no information on its own, and thus was not relevant to our analysis.

   We adopted a shallow parsing approach for extracting information from the NL text. Shallow parsing is a lightweight text analysis method that performs a (potentially) partial analysis of the linguistic structures in a text. We used the Cico domain-based parser [Ambriola & Gervasi 1997], a tool based on fuzzy matching of sentence fragments to templates, with a rule set specifically developed for the language used in the NCS specification. Building this rule set required no more than two days of work by one of the authors. Given the highly specialized language used in the document, 30 rules (in addition to the generic rules already provided by the tool) were sufficient to obtain complete parsing of three different revisions of the document.

2. *Selecting desirable properties to check.* Given that most of the requirements dealt with assigning certain values to specific output lines upon the occurrence of some event, we decided to perform "black-box" validation of the requirements. In particular, we were interested in the possible values that each output line could assume. Some of the properties we selected at this stage (those that uncovered problems in the specification) are presented later. Formally, input and output lines were described in terms of associated data items, whose value could change outside system control (for input lines) and whose assignment caused side effects (for output lines).

3. *Defining models for checking selected properties.* All the properties we defined could be checked on the four models described below:
   - the kind of data item (KIND) model, distinguishing constant values from internal variables and I/O items. Formally, $\forall\ d \in$ DataItems, kind(d) is either CONST (a constant value), FLAG (an internal variable) or IO (an input/output line).
   - the default values (DEFVAL) model, showing only the default or initialization value of data items, as declared in the requirements. Formally, $\forall\ d \in$ DataItems, defval(d) is either UNDEF (no initialization value was specified by the requirements) or a set of specific literal values.[2]
   - the value space (VALSPACE) model, collecting all the assignments described in the requirements to determine the space of all the possible values for a data item. Formally, $\forall\ d \in$ DataItems, valspace(d) is the set of all values whose assignment to d or whose comparison with the value of d is mentioned in the requirements.
   - the event-condition-action table (ECATAB) model, collecting all the possible actions of the system, together with the conditions and events that cause their execution. Formally, $\forall\ r \in$ Requirements, ecatab(r) = { < *events*, *conditions*, *actions* > } where *events* and *conditions* are predicates on the

[1] As an extreme example, some of the functions described had to behave differently depending on the orbital position of the space station, as expressed by an orbit diagram included in the text. It is typical of lightweight formal methods (and thus of partial validation) that none of the models really need to "understand" the details behind such a diagram, and in our case study we could simply treat a change in orbital position as an unexplained event.

[2] No initializations with non-literal values were specified in our requirements, but if present, they could have been treated as an assignment to be performed unconditionally upon a "Boot" event in the ECATAB model. Notice also that although defval(d) is specified as a set, a double initialization with different values would have been regarded as an error in the requirements ($\forall$ d $\in$ DataItems, #defval(d)=1 was one of the desirable properties). In our case study, this property was never violated, and defval(d) always resulted in a singleton.

value of members of DataItems, while *actions* is a set of actions (either assignments to members of DataItems or the special actions "Acquire *d*", signifying the assignment to *d* of a value read from an input device, and "Reject *d*", indicating the rejection of a command according to the bus protocol, with $d \in$ DataItems), as specified by requirement *r*.

We used the Circe environment [Ambriola & Gervasi 1997] to provide tool support for the production phase. Circe is a Web-based environment for the automated analysis of requirements written in natural language. The environment supports the extraction of models from the requirements, their validation, and the collection of metric data about the requirements document, the system described in it and about the requirements writing process itself.

4. *Pre-processing the requirements document.* The text of the requirements was simply copied and pasted from the original Word document into our tool, and needed very little manual preprocessing (e.g., commenting out section titles and changing enumerated lists to bulleted lists). Such preprocessing could have been performed automatically if the document size had required it.

5. *Parsing of the NL text.* The parsing technique we adopted required that a glossary be defined containing domain-specific terms. This task was accomplished by populating the glossary with:
   - the names of the various data items from the input/output tables included in the specification document,
   - the name of the system itself ("NCS"), and
   - a few other names that were used in the requirements (even though they were not declared as input or output data or command names).

   The parsing process in itself provided a *language* validation of the requirements. No spelling or syntactic errors were found, supporting our assumption that the requirements were of good *syntactic quality* [Krogstie *et al.* 1995] with respect to the language defined by our parsing rules. Obviously, only a partial assessment of the *semantic quality* defined in the same work was performed, by formally validating the models as described below.

6. *Building models.* The task of building the four models defined in step 3 was carried out by a small number of *modelers*, i.e. plug-ins of Circe's modular architecture. The logic needed to build these models, implemented as additional modules of the existing tool, was less than 100 lines of AWK [Aho *et al.* 1988] code.

7. *Checking that the models satisfy chosen properties.* The properties we selected were analyzed by a number of specialized *validators* (also implemented as plug-ins for Circe), each only a few lines of code long. Many of the chosen properties, mostly concerning "obvious" completeness and consistency criteria, were never violated, and are not reported here. Violations that led to the identification of real problems in the specification are discussed in some detail in the following.

4.2. Findings of the validation

As mentioned above, the VALSPACE model collected all the values mentioned in the requirements as assignable to each data item, either as default values or by explicit statements. One of the properties we wanted to verify on this collection was simply that every non-constant data item had more than a single possible value, or

$$\forall\ d \in \text{DataItems},\ \text{kind}(d) \neq \text{CONST} \Rightarrow \#\text{valspace}(d) \geq 2$$

We found six different data items (listed in Table 1) that did not satisfy this simple property. Closer inspection triggered by this finding revealed that several data items whose labels started with "ACS N1-2" were synonyms to other data items whose names started with "ACS N1S2" — with the N1-2 label (probably) left over from previous releases. We had originally interpreted these as distinct data items, as the SRS document contained many other distinct data items with only slight variations in their names.

Table 1: Data items failing VALSPACE validation.

Data item name (d)	*valspace(d)*	*Reason*
“ACS N1-2 Cabin Pressure Lower Limit Warning State”	{TRUE}	Synonym with “ACS N1S2 Cabin Pressure Lower Limit Warning State”.
“ACS N1S2 Cabin Pressure Upper Limit Warning State”	{FALSE}	Synonym with “ACS N1-2 Cabin Pressure Lower Limit Warning State”.
“High pressure warning level alarm”	{“return to normal”}	Alarms **issue**d, not set; also synonym with the “Upper limit” alarm.
“Low pressure warning level alarm”	{“return to normal”}	Alarms **issue**d, not set; also synonym with the “Lower limit” alarm.
“Upper limit warning level alarm”	{“return to normal”}	Alarms **issue**d, not set; also synonym with the “High pressure” alarm.
“Lower limit warning level alarm”	{“return to normal”}	Alarms **issue**d, not set; also synonym with the “Low pressure” alarm.

We also had to revise our understanding of the alarm handling by the system. The document used the wording ***issue** an alarm* to indicate entrance into an alarm state, and ***set** alarm to “return to normal”* to indicate exiting from an alarm state. We had originally taken these as unrelated operations. Inspection of the relevant section of the SRS document confirmed that issuing an alarm should be interpreted as ***setting** an alarm to “in alarm”* — a change in interpretation that was reflected by a simple update of the parsing rule for ***issue*** in our rule set.

Another issue related to alarms was that the SRS referred to the same alarm in different ways. For example (letters refers to requirements in the SRS):

d. (The NCS shall) *issue a warning level alarm (message: “Node 1 Cabin Pressure Lower Limit Warning Violation”)*
e. (The NCS shall) *set the lower limit warning level alarm to “return to normal”*
j. (The NCS shall) *set the low pressure warning level alarm to “return to normal”*

There was potential confusion in the first requirement (***d***) between the identity of an alarm (a system design issue) and the associated warning message (a user interface issue). Again, inspecting the document with this finding in mind confirmed that the three different designations above were indeed referring to the same entity. The same problem also occurred in three other requirements, and correcting it reduced the total number of alarms from six to two. Tabulation of alarms, as for other I/O items, may have avoided such confusion.

Once we resolved issues in the VALSPACE model, the DEFVAL model became amenable to further analysis. So, we tested the model for the property:

$$\forall\ d \in \text{DataItems},\ \text{defval}(d) \neq \text{UNDEF}$$

which states that each data item should have a declared default value. Four items did not satisfy this property: the two alarms discussed above, the reported Cabin Pressure value, and the Confirmation Command Rejection Indicator (a flag of the command bus protocol). We already knew that the default state for all alarms was “return to normal” (i.e. no alarm), but the other two data items could potentially provide false information if read before the first assignment.

A better understanding of these potentially dangerous conditions was gained by looking at the ECATAB model. This model, inspired by the Event tables of SCR [Heitmeyer *et al.* 1998], shows which actions (A) the system performs when a certain event (E) occurs, and which conditions (C) must hold for the actions to be performed. Since the Cabin Pressure Monitoring function was not described in terms of states, we assumed the system to always be in a “Normal” state. The ECATAB model synthesized by our tool from the NL text of the requirements is shown in Table 2, where we substitute abbreviations for the very long data and command names used in the SRS document. Table 2 uses an SCR-like notation: for each requirement *r*, and for each triple in ecatab(*r*), conjuncts in the *events* predicate are marked with “@T” (read: “becomes true”), conjuncts in the *conditions* predicate are marked with “T” (read: “is true”) or “F” (read: “is false”) if they are negated, and elements of the *actions* set are explicitly listed under the Actions heading.

Table 2: ECATAB model.

Req		Sample Cycle (1Hz)	FDIR = Enabled	P<LL x3 s.c.	P>LL x3 s.c.	P>UL x3 s.c.	P<UL x3 s.c.	D.FDIR	D.FDIR CC	CCREQ = Enabled	E.FDIR	Actions
a		@T										Acquire P
d			T	@T								LLWS:=TRUE LLW:=IN_AL
e					@T							LLWS:=FALSE LLW:=RTN
f			T			@T						ULWS:=TRUE ULW:=IN_AL
g							@T					ULWS:=FALSE ULW:=RTN
h								@T				CCREQ:=ENABLED CCREJ:=FALSE
i	1								@T	T		FDIR:=DISABLED CCREQ:=DISABLED
	2								@T	F		Reject D.FDIR CC CCREJ:=TRUE
j											@T	CCREQ:=DISABLED FDIR:=ENABLED LLWS:=FALSE ULWS:=FALSE LLW:=RTN ULW:=RTN

This table passes the usual consistency checks; e.g., disjointness and coverage, under the customary one-input assumption[3] [Heitmeyer *et al.* 1996] — i.e., assuming that exactly one variable changes value between the execution of two sets of actions . Given a set of actions A, we define its *write-set*, W(A), as the set of data items modified by the actions in the set, i.e.

$$W(A) = \{v \mid v{:=}value \in A\} \cup \{v \mid \text{Acquire } v \in A\}.$$

Using Table 2, we identified the conditions under which the two variables above, Cabin Pressure (P) and Confirmation Command Rejection Indicator (CCREJ), are not initialized (by simply inspecting the rows r_i that satisfy P $\in$ W(actions(r_i)) and CCREJ $\in$ W(actions(r_i)), respectively). P only occurs in the row for requirement ***a***, so P is not initialized until the first sampling cycle occurs. Depending on the actual subsequent implementation, a first sample could be taken immediately upon startup — before even listening to the bus, or it could be delayed for as much as a second — during which time other components of the system reading the pressure value could obtain erroneous (random) results. CCREJ on the other hand is modified only by actions in rows ***h*** and ***i***$_2$, so it is not initialized until the first Disable FDIR or the Disable FDIR Confirmation Command (issued while CCREQ is DISABLED, which is its default value) comes up the bus. Since CCREJ is part of the bus protocol, we have to assume that this behavior is documented and does not (currently) constitute a problem.

If the one-input assumption cannot be guaranteed, we can still identify potentially dangerous conditions by examining incompatible requirements. Two action sets A and B are *incompatible* if they assign different values to the same variable, i.e. $\exists\, v \in W(A) \cap W(B)$ s.t. ($v{:=}a \in A \wedge v{:=}b \in B \wedge a{\neq}b$) $\vee$ (Acquire $v \in A{\cup}B$) (for our purposes, "Acquire v" assigns to v an unknown value, potentially incompatible with any other value). In order to avoid conflicting assignments to the same data item, the conjunction of the events and conditions of rows with incompatible action sets must always be false, i.e.

$$\forall\, r_1, r_2 \in \text{Requirements},\ \forall <evt_1, cond_1, act_1> \in \text{ecatab}(r_1),$$
$$\forall <evt_2, cond_2, act_2> \in \text{ecatab}(r_2),$$
$$\text{incompatible}(act_1, act_2) \Rightarrow \neg(cond_1 \wedge cond_2 \wedge evt_1 \wedge evt_2)$$

In Table 2, the following rows are incompatible: (***d***,***e***), (***d***,***j***), (***f***,***g***), (***f***,***j***), (***h***,***i***$_1$), (***h***,***i***$_2$), (***h***,***j***), (***i***$_1$,***j***), but only the first and third pair satisfy the above property. In fact, they can be discounted by simple arithmetic (for example, (***d***,***e***) represents the case when the Pressure becomes at the same time higher and lower than the admissible Lower Limit). The second and fourth cases are worth flagging as potentially problematic. They represent the case when *enabling* the FDIR feature continuously (i.e. enabling the reporting of alarms) can actually *prevent* the alarms from firing, depending on the order in which events are checked by the code. This problem can be avoided by checking that FDIR = Enabled is FALSE as a condition for performing the actions in requirement ***j***. The remaining cases formalize bus protocol violations, and can be ignored for a serial bus.

[3] This assumption holds for events triggered by the reception of commands, due to the serial nature of the 1553 bus used to carry them. We do not know if the one-input assumption actually holds between bus events (command reception) and timer events (sampling cycle, testing read pressure values). In a complete study, the assumption should be verified by inspection of the actual code or detailed design document for our subsystem. The VALSPACE model shows that FDIR and CCREQ can be either ENABLED or DISABLED only, so that checking for either value is sufficient. The system exhibits a certain hysteresis, simply maintaining the previous state for boundary cases where Pressure exactly equals the Upper or Lower Limit.

5. Lessons learned

The original aim of this study was not the validation of the requirements specification described, but to experiment with the use of lightweight formal methods in an RE process based on NL requirements and inspections. The total effort spent on initial setup, done once only by one of the authors – familiar with the parsing and modeling environment but completely ignorant of the problem domain[4], was less than three working days. Subsequently, parsing the sample specification took less than 10 seconds, while generating and validating the various models took between 0.5 and 2 seconds on a desktop PC. No training was required for the requirements writers (we used the original document *verbatim*), and little explanation of how to interpret the results of the process is needed by a V&V team.

Such low computational and human costs strongly suggest that our techniques could be used successfully *during* requirements writing and evolution[5]. We believe that the application of these techniques could also help in tracking the quality of a specification *between* full releases and inspections, thus providing finer grain monitoring of the RE process and to some extent validating changes that are requested and implemented.

While the inconsistencies we identified in our analysis currently appear to be non-critical, unfortunate experiences (such as the Ariane 5 disaster [Nuseibeh 1997]) have shown us that risk assessments can change as requirements and other circumstances change. The availability of lightweight formal methods to identify and track inconsistencies in natural language requirements documents is therefore invaluable.

It is particularly interesting to note that a number of the problems we discovered via lightweight validation (e.g., the need for alarms tabulation, the issue/set terminological problem, the need for unequivocal alarm designations and the separation between design and user interface issues) were also *independently* discovered and corrected by NASA's IV&V team. In fact, the revision of the NCS specification immediately following the three revisions we used in our case study, was changed in line with our findings. The latest revision also included some evolutionary changes that were not prompted by errors in the previous releases, but still presented the errors related to uninitialized and to inconsistently named data items that we discussed above. In light of the partiality of the validation performed by our lightweight methods, we regard the fact that no error was discovered by the IV&V team that had gone undetected by our study as merely a casual — albeit comforting — occurrence.

[4] This can be an advantage in trying to identify "obvious" errors [Berry 1995].

[5] For very large documents, differential parsing and model updating can be used instead of re-parsing the entire document each time. In this case, parsing time drops to 1-2 seconds on average, and is influenced only by the amount of changes in the document between validation iterations, and not by the document size. In our experience, the amount of changes between revisions tends to remain constant, and so is parsing time.

6. Related Work

The idea that requirements can be analyzed in an automatic fashion in order to identify and possibly correct several kind of errors has attracted much attention. Most proposals call for a formal specification of the requirements to begin with. [Heitmeyer *et al.* 1998] present a method based on the Four-Variables Model by Parnas and Madey, discuss the design of a prototype tool, and give conditions under which the consistency checks can be complete. However, in their proposal requirements must be expressed in the formal SCR notation (based on finite state machines and event/conditions tables). Similarly, [Jackson & Damon 1996] in their validation tool Nitpick assume that requirements are expressed in a subset of Z, and [Reubenstein & Waters 1991] call for "natural language-like" requirements written in LISP. Validation via model checkers like Nitpick or SPIN [Holzmann 1995] can provide a high degree of confidence, but the formalization step in itself is prone to errors, so some connection to informal requirements is often sought.

[Duffy *et al.* 1995] try to integrate formal and informal representations of the requirements by mandating that *both* must be stated side-by-side; analysis is then performed on the formal version. However, equivalence between the two representations of the same requirement is only assumed, and no guarantee is given of their actual correspondence. So, effectively, their proposal amounts to annotating a formal requirements document with abundant natural language comments. Others (e.g., [Dalianis 1992]) propose to *generate* a natural language paraphrase of a formal requirements document to help interaction with the customer and other non-technical participants in the software development process. This approach is the exact dual of the one we presented in the present paper, based on parsing of natural language requirements. Both techniques can be used in the same project, and actually complement each other well. Other work related to the techniques we used in the case study include [Rolland 1992] – where parsing techniques are used to extract a conceptual model from natural language requirements, and [Macias & Pulman 1993] – in which a strict syntax is imposed on the requirements to ensure syntactic quality.

7. Conclusions and Future Work

In this paper we have presented a structured approach for validating natural language requirements. We applied our approach to an industrial case study, which served to demonstrate the feasibility and benefits of lightweight formal methods in this context. The low cost of our approach, both in terms of human training and of computational resources needed, makes it particularly well suited for the initial introduction of formal methods in an organization.

Although we examined a single requirements document, the techniques we developed for the case study can be reused to validate subsequent releases of the same document, and indeed can be applied to other projects making use of a similar document style. So, for example, the effort spent defining parsing rules need not be duplicated in subsequent projects by the same organization.

If necessary, more complex models could be constructed. For example, our ECATAB model could be extended to a full SCR model. Frequent lightweight validation could be performed on the simpler models (like ECATAB), while less frequent, more complete analysis could be performed on the more costly models.

We expect that embedding lightweight formal methods into a requirements engineer's everyday development environment would provide substantial productivity benefits. For example, a requirements management tool could be adapted to provide a validation function as part of its analysis capabilities. We are in the early stages of cooperating with a commercial firm for this purpose.

Acknowledgement

We would like to thank Khalid Lateef and John Hinkle of the NASA IV&V Facility, West Virginia, USA, for providing us with the case study and for their valuable feedback and discussions. Thanks also to Jack Callahan of NASA/WVU for many useful insights into NASA V&V procedures. Bashar Nuseibeh would like to acknowledge NASA's financial support under NASA Cooperative Agreement #NCC 2-979, and the UK EPSRC for the MISE (GR/L 55964) and VOICI (GR/M 38582) projects. Vincenzo Gervasi would like to acknowledge the financial support of RENOIR.

References

[Aho *et al.* 1988] A. V. Aho, B. W. Kernighan and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Ma., 1988.

[Ambriola & Gervasi 1997] V. Ambriola and V. Gervasi, "Processing Natural Language Requirements", *Proceedings of the 12th IEEE Conference on Automated Software Engineering,* 36-45, November 1997.

[Berry 1995] D. M. Berry, "The Importance of Ignorance in Requirements Engineering", *Journal of Systems and Software*, 28(2): 179-184, February 1995.

[Dalianis 1992] H. Dalianis, "A method for validating a conceptual model by natural language discourse generation", (In) *Advanced Information Systems Engineering*, Springer, LNCS 593, 1992.

[Duffy *et al.* 1995] D. Duffy, C. MacNish, J. McDermid and P. Morris, "A framework for requirements analysis using automated reasoning", *Lecture Notes in Computer Science*, 932, 1995.

[Easterbrook *et al.* 1998] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling*", IEEE Transactions on Software Engineering,* 24(1): 4-14, January 1998.

[Fabbrini *et al.* 1998] F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi and S. Ruggieri, "Achieving Quality in Natural Language Requirements", *Proceedings of the 11th International Software Quality Week,* San Francisco, May 1998.

[Feather 1998] M. S. Feather, "Rapid Application of Lightweight Formal Methods for Consistency Analyses", *IEEE Transactions on Software Engineering*, 24(11): 949-959, November 1998.

[Goldin & Berry 1997] L. Goldin and D. M. Berry, "Abstfinder: A Prototype Natural Language Text Abstraction Finder for Use in Requirement Elicitation", *Automated Software Engineering Journal*, 4(4): 375-412, October 1997.

Hietmeyer *et al.* 1996] C. Heitmeyer, R. D. Jeffords and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications", *ACM Transactions on Software Engineering and Methodology*, 5(3):231-261, July 1996.

[Heitmeyer *et al.* 1998] C. Heitmeyer, J. Kirby, B. G. Labaw and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements", *Proceedings of 10th Annual Conference on Computer-Aided Verification (CAV'98),* Vancouver, Canada, 1998.

[Hesketh *et al.* 1998] J. Hesketh, D. Robertson, N. Fuchs and A. Bundy, "Lightweight Formalisation in Support of Requirements Engineering", *Automated Software Engineering*, 5(2):183-210, April 1998.

[Holzmann 1995] G. J. Holzmann, “Proving properties of concurrent systems with SPIN”, *Lecture Notes in Computer Science,* 962, 1995.

[Jackson & Damon 1996] D. Jackson and C. A. Damon, “Elements of style: Analyzing a software design feature with a counterexample detector”, *IEEE Transactions on Software Engineering*, 22(7):484-495, July 1996.

[Jackson & Wing 1996] D. Jackson and J. Wing, “Lightweight Formal Methods”, *IEEE Computer,* 29(4): 21-22, April 1996.

[Kemmerer 1990] R. A. Kemmerer, “Integrating Formal Methods into the Development Process” *IEEE Software*, 7(5): 37-50, September 1990.

[Krogstie *et al.* 1995] J. Krogstie, O. I. Lindland and G. Sindre, “Towards a deeper understanding of quality in requirements engineering”, *Proceedings of the 7th International CAiSE Conference,* 82-95, Springer, LNCS 932, 1995.

[Murphy *et al.* 1995] G. C. Murphy and D. Notkin, “Lightweight Source Model Extraction”, *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 116-127, October 1995.

[Macias & Pulman 1993] B. Macias and S. G. Pulman, “Natural Language Processing for Requirements Specification”**,** (In) *Safety-critical Systems*, 57-89, Chapman and Hall, 1993.

[NASA 1997] NASA/Boeing, “Software Requirements Specification for the NCS MDM CSCI*”, International Space Station Document S684-10174,* April 1997.

[Nuseibeh 1997] B. Nuseibeh, “Ariane 5: Who Dunnit?”, *IEEE Software*, 14(3): 15-16, May 1997.

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

[Rolland 1992] C. Rolland, “A Natural Language Approach for Requirements Engineering”, (In) *Advanced Information Systems Engineering*, Springer, LNCS 593, 1992.

[Russo *et al.* 1998] A. Russo, B. Nuseibeh and J. Kramer, “Restructuring Requirements Specifications: a case study”, *Proceedings of 3rd IEEE International Conference on Requirements Engineering (ICRE’98)*, 51-60, Colorado Springs, USA, April 1998.

[Russo *et al.* 1999] A. Russo, B. Nuseibeh and J. Kramer, “Restructuring Requirements Specifications”, *IEE Proceedings: Software,* 144(1):44-53, February 1999.