SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2002; **32**:113–133 (DOI: 10.1002/spe.430)

# Lightweight validation of natural language requirements

Vincenzo Gervasi[1] and Bashar Nuseibeh[2,*,†]

[1]*Dipartimento di Informatica, Università di Pisa, I-56125 Pisa, Italy*
[2]*Computing Department, The Open University, Milton Keynes MK7 6AA, U.K.*

**SUMMARY**

**In this paper, we report on our experiences of using lightweight formal methods for the partial validation of natural language requirements documents. We describe our approach to checking properties of models obtained by shallow parsing of natural language requirements, and apply it to a case study based on part of a NASA specification of the Node Control Software on the International Space Station. The experience reported supports 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 Independent Validation and Verification Facility in a subsequent version of the same document, and others that were not discovered. The paper describes the techniques we used, the errors we found and reflects on the lessons learned. Copyright © 2001 John Wiley & Sons, Ltd.**

KEY WORDS: 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 [1,2]. In the context of requirements engineering (RE), we use the term 'lightweight formal methods' to characterize 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 [3]. They do not require a

*Correspondence to: Bashar Nuseibeh, Computing Department, The Open University, Milton Keynes MK7 6AA, U.K.
†E-mail: B.A.Nuseibeh@open.ac.uk

Contract/grant sponsor: NASA; contract/grant number: #NCC 2-979
Contract/grant sponsor: UK EPSRC (MISE); contract/grant number: GR/L 55964
Contract/grant sponsor: UK EPSRC (VOICI); contract/grant number: GR/M 38582
Contract/grant sponsor: MURST (AI:IA project)
Contract/grant sponsor: EU (RENOIR and PROMOTER 2 projects)

Copyright © 2001 John Wiley & Sons, Ltd.

*Received 14 August 2000*
*Revised 26 September 2001*
*Accepted 26 September 2001*

commitment to translate entire (informal) requirements documents into formal ones, nor to maintain formal and informal versions of specifications in parallel [4]. 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 (such as [5] where lexical analysis is used to find abstractions in unstructured and uninterpreted text), to design support systems [6] and reengineering applications on existing code [7]. Others have studied the application of natural language (NL) understanding techniques to the automatic extraction of models from NL requirements [8–13]. 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. It is not our intention to suggest that the particular techniques used in this study can be applied to any kind of RE process. Rather, in the vein of [6], we present the results of the study as evidence to support our position that lightweight methods can be profitably used in RE processes that deploy NL requirements.

The paper is structured as follows. We begin in Section 2 by introducing our general approach to the lightweight validation of NL requirements. The approach includes a set-up phase to adapt the general schema to the particular needs of a specific requirements process, and a production phase, where requirements are repeatedly checked during the evolution of a requirements document. We then introduce our case study in Section 3, and provide some background on its origin and its significance. In Section 4 we describe the application of our framework to the case study and provide a complete example of the kind of processing that is applied to NL requirements for the purpose of validating them with respect to a certain set of desired properties. Our findings from the case study are discussed in Section 5, where the validation technicalities are also discussed in more detail. We reflect on the lessons we learned in Section 6, and discuss the applicability of our experiences in different—and, particularly, in industrial—contexts. A short survey of related work and a discussion of future work conclude the paper.

## 2. VALIDATING NL REQUIREMENTS

The term *verification* has traditionally been used to designate checking that a software system conforms to its specification, and the term *validation* to designate checking that the specification captures the actual needs (or the expectations) of its customers [14]. The focus of this paper is on the validation of software system requirements.

The term *validation* is also used to indicate checking that a model (of a software system) satisfies certain internal consistency properties, under the assumption that internal consistency of a software system is always part of the customer's expectations. In this sense, validation of a requirements document is of necessity partial. It is impossible to guarantee that a specification satisfying any fixed set of properties on the underlying models will cover all the (often unexpressed) user requirements [15].

Copyright © 2001 John Wiley & Sons, Ltd.

In this paper, we treat validation as a decision problem: given a model of a software system and a set of properties, we want to know whether or not the model satisfies those properties. If not, it is desirable to provide one or more counter-examples showing how a property was violated. Notice that both the model and the set of properties are arbitrary, and chosen case by case according to which characteristics are more important to the customer. As an example, in a hard real-time system, exact timings can be very important and the precision of results from computation may have to be sacrificed for the timeliness of the result; on the other hand, an accounting system must be exact and the time needed to complete a computation is only of secondary concern. In these cases, different user needs‡ lead to different views of validation, with different properties required to be satisfied by different models.

The aim of our study is not the validation of the particular requirements specification described in Section 3, but rather to experiment with the use of lightweight formal methods in an RE validation process based on NL requirements and inspections.

Our approach is structured into two parts: a *set-up phase* and a *production phase*. The two phases, and their relationships, are depicted in Figure 1. The set-up phase includes the following activities.

1. *Defining a style a structure and a language for the requirements document*. This step can be undertaken 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, as in [16]), 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 in this paper).
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 against 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 analysable structure the information needed to check the property. For example, a connection property among system components can be checked against a model describing all the communications among system components.

Once the set-up phase has been completed, the production phase (below) 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 the following.

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 analysable representation of the semantic content of the text. Again, parsing can be (and usually is) partial, to help reduce the cost of validation, as long as this 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

‡Or even different priorities for similar needs.

Copyright © 2001 John Wiley & Sons, Ltd. 

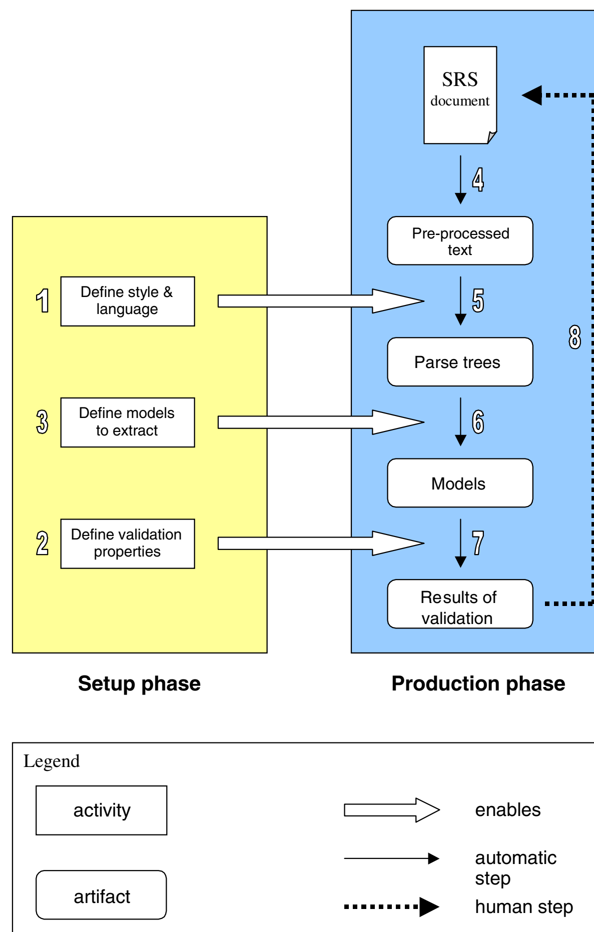


Figure 1. Set-up phase and production phase in our approach.

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 of and the reason for a failure (i.e. about the circumstances in which a validation property was

Copyright © 2001 John Wiley & Sons, Ltd. 

violated). Like the counterexamples provided by other formal methods, this information helps the requirements engineer to identify and fix errors that cause violations.

The process above offers a number of advantages in an industrial setting. Steps 1–3 are reusable across projects, as individual organizations tend 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.

## 3. 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 [17]. The choice of this particular document was appealing because we assumed it to be of high quality (being the twelfth release of those requirements, and subject to many inspections and revisions), and because parts of it had already been analysed using different techniques, in related studies [3,18,19].

The document, 250 pages long, is written mainly in narrative English, with several tables and the occasional schematic diagram interspersed in the text. It 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 and 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. This second part (called ‘engineering requirements’) constitutes the official (and definitive) specification of the NCS, while the narrative part is provided as explanatory material. Cross-references and references to external documents are used in places to mandate standard compliance and uniform behaviour among different functions.

The three-page fragment (Section 3.2.2.1.3 in [17]) we chose to analyse 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. Operating limits can be changed during the system’s operation. The NCS interfaces with the physical world through input lines connected to sensors and output lines connected to actuators; interaction with the rest of the system is through a shared serial bus carrying commands and state information. Precise timings are given for most (but not all) of the system’s operation, e.g. to ensure that operating limits are exceeded across a certain number of samplings before declaring an alarm and to ensure that alarms are issued in a bounded time when a problem is discovered.

Moreover, certain procedures relevant for the bus protocol are specified. For example, commands that are invalid in particular circumstances are to be ignored, but a rejection indication must be provided to the originator of the command. Similarly, commands that may be dangerous are accepted only after a separate specific confirmation command is received by the subsystem.

Most other subsystems in the NCS are comparable in structure and complexity to the Cabin Pressure Monitoring function we analyse in this case study, while a few are considerably more complex. Many

Copyright © 2001 John Wiley & Sons, Ltd.

subsystems could be validated in isolation using the techniques we describe in the next sections. However, we do not explore the issue of validating the entire system in this work. Given the substantial size of the NCS, even the simplest integration tests would have required more domain knowledge than we possessed.

## 4. APPLICATION OF THE 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 in Section 2. Since—due to logistic considerations—we had to work alongside NASA's standard verification and validation process, we only ran a single iteration of our production phase. Also, we analysed three major revisions of the requirements document. The first version was provided to us at the beginning of the study, while two other subsequent versions—developed independently from our preliminary findings on the first version—were released before its end. Our analysis was performed in 'batch mode': the requirements were provided to us in their final form; no interaction with the developers took place during our analysis. This is not the best possible setting for lightweight validation, which is actually better suited for interactive use (thus in a sense performing continuous validation) *during* requirements evolution, possibly after every minor change. However, this unfavourable setting offered the opportunity to compare our findings with those of a traditional validation and verification (V&V) process as performed by NASA (mostly inspection). We describe the results of this comparison in Section 6.

Our approach offers more flexibility in writing and analysing requirements than the eight steps listed in Section 2 suggest. Indeed, we expect that the approach be instantiated to deal with the peculiarities of each different organization. Among different instantiations, some reuse of language, models and properties can be achieved. The extent of this reuse may be almost total for similar projects of the same organization, may become smaller, but still substantial, for different organizations in the same industry or may be absent or minimal in cases where two organizations operate in different industries. In our case study, we simulated the initial introduction of our approach into an organization. This is the most extensive instantiation that can be made, since no reuse is possible. First, we instantiated the steps of the set-up phase as follows.

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* [20]. 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[§] 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.

[§]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 external event.

Copyright © 2001 John Wiley & Sons, Ltd. 

We adopted a shallow parsing technique 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 [21], 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. In domain-based parsing, parse trees do not describe the syntactical structure of a sentence, but rather its content according to structures that are significant in the domain. Building a rule set for the domain of our case study 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 (i.e. we only validated the externally-observable behaviour of the system). In particular, we were interested in the possible values that each output line could assume. For example, we wanted to be sure that output lines would never be left in undefined states, due to missing initializations or to requirements conflicting on their values. We also wanted to be sure that the system behaviour was defined even for unusual combinations of values from the input lines and commands from the command bus. Some of the properties we selected at this stage (those that uncovered problems in the specification) are presented in more detail 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 in the previous step could be checked against 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[¶].
    - 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, $\forall r$

[¶]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.

Copyright © 2001 John Wiley & Sons, Ltd.

$\in$ Requirements, ecatab$(r) = \{\langle events, conditions, actions\rangle\}$ where *events* and *conditions* are predicates on the 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 $r$. Intuitively, ecatab$(r) = \{\langle e, c, a\rangle\}$ can be read as 'Requirement $r$ specifies that, upon occurrence of the event $e$, if the condition $c$ is satisfied, the actions in $a$ shall be executed'.

We used the Circe environment [21,22] to provide tool support for the production phase. Circe is a Web-based, component-based environment for the automated analysis of requirements written in natural language (using Cico for the parsing stage). 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. It has been used as a requirements writing support system in a number of industrial and academic projects, but its application to the analysis of pre-existing requirements is novel. Using Circe, steps 4 to 7 of our approach were implemented as follows.

4. *Pre-processing the requirements document*. The text of the requirements was simply copied and pasted from the original Microsoft 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 manually 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 implicitly 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* [23] 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 automatically by a small number of *modellers*, i.e. software components that are part 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 [24] code.
7. *Checking that the models satisfy chosen properties.* The properties we selected were analysed by a number of specialized *validators* (also implemented as components for Circe), each consisting of a few lines of code only. Many of the chosen properties, mostly concerning 'obvious'

Copyright © 2001 John Wiley & Sons, Ltd. 

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 Section 5.

*Example.* To illustrate the operations in steps 4–7 above, we now present an example of how a single requirement from the SRS was actually processed. Consider the following requirement, taken *verbatim* from the original document:

d. If the ACS N1S2 Cabin Pressure FDIR State is equal to ENABLED, then upon receipt of ACS N1S2 Cabin Pressure HW Data less than the Node 1 Cabin Pressure Lower Limit (initial: 13.9 psia) for 3 consecutive acquisitions, the NCS shall within 1.1 seconds:
   (1) Set the ACS N1S2 Cabin Pressure Lower Limit Warning State (initial value: FALSE) to TRUE, and
   (2) Issue a warning level alarm (message: "Node 1 Cabin Pressure Lower Limit Warning Violation") in accordance with the section on "annunciate alarms".

Pre-processing of this text (step 4) only required minimal intervention, mainly to substitute enumerated list markers with a different markup and to change the format of the text from the original one (Rich Text Format) to a simple ASCII version. After pre-processing, the requirement above appeared as follows:

```
If the ACS N1S2 Cabin Pressure FDIR State is equal to ENABLED, then
upon receipt of ACS N1S2 Cabin Pressure HW Data less than the Node 1
Cabin Pressure Lower Limit (initial: 13.9 psia) for 3 consecutive
acquisitions, the NCS shall within 1.1 seconds
- Set the ACS N1S2 Cabin Pressure Lower Limit Warning State (initial
value: FALSE) to TRUE, and
- Issue a warning level alarm (message: ``Node 1 Cabin Pressure Lower
Limit Warning Violation'') in accordance with the section on
``annunciate alarms''.
```

The parsing process of step 5 analyses this pre-processed text, producing the parse trees shown in Figure 2 (in which some data item names have been abbreviated for clarity of presentation). Recall that in domain-based parsing, parse trees describe the content of the requirement according to structures that are significant in the domain and not according to traditional syntactical structures. For example, the root of the main tree describes a causal dependency (DEPC) between a condition—as expressed by the relational operator (RELOP) 'equals' between Cabin Pressure FDIR State and ENABLED—and an action, starting at the DEPT node. That action is itself a temporal dependency (DEPT) between the conditional receipt (CONDRECEPIT) of a Cabin Pressure HW Data less than the Cabin Pressure Lower Limit lasting (EVTFORSPAN) for at least three sampling times (TSPAN). As soon as the dependency is satisfied, the block of actions starting at the DOOP node, which corresponds to points (1) and (2) in the original requirement, is executed. Notice also how incidental information like declarations of default values has been parsed into distinct trees. Details of this parsing technique can be found in [22].

Models are built from such a forest of parse trees (step 6) by the modellers developed in the set-up phase. Modellers can extract information directly from parse trees or from models synthesized by other

Copyright © 2001 John Wiley & Sons, Ltd. 

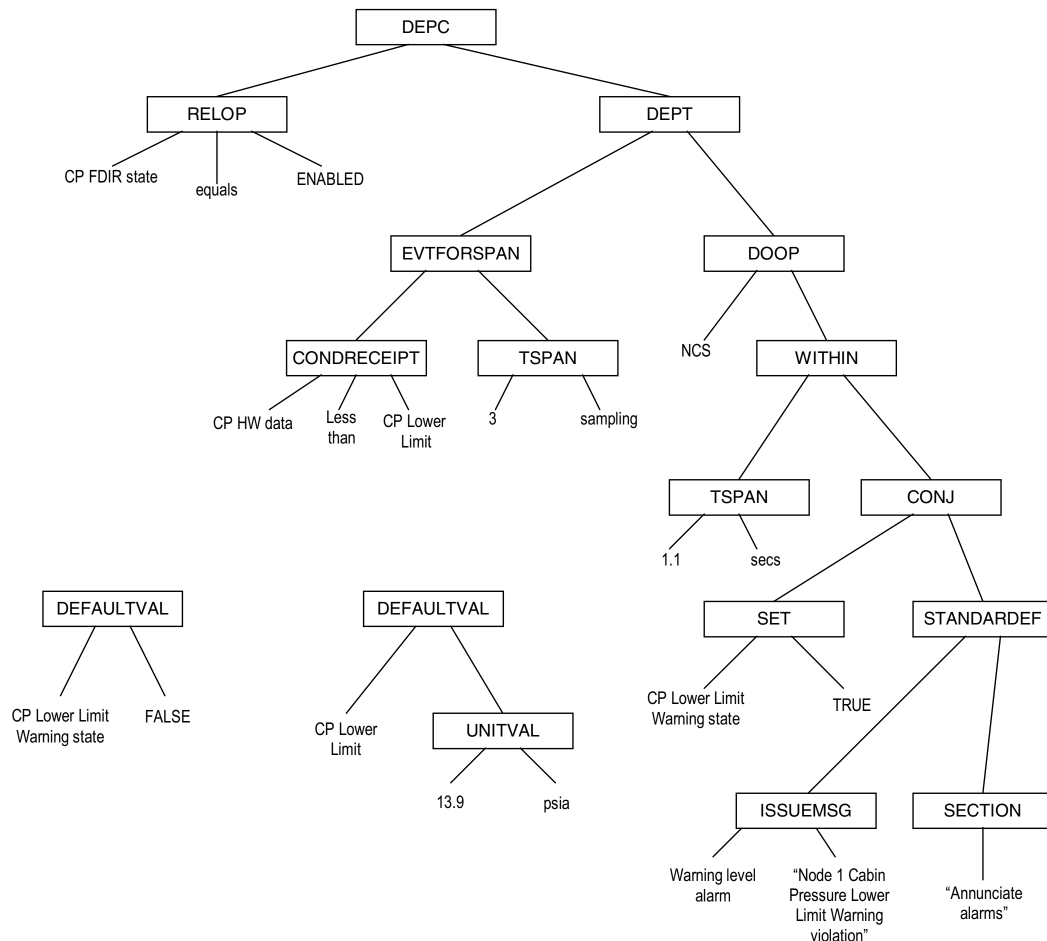


Figure 2. Parsing trees for the sample requirement.

modellers. In our example, the DEFVAL and VALSPACE models are populated directly from the parse trees giving‖

$$\text{defval(CP Lower Limit Warning state)} \supseteq \{\text{FALSE}\}$$
$$\text{defval(CP Lower Limit)} \supseteq \{13.9\ \text{psia}\}$$

‖We use the notation *model* $\supseteq$ {*value*} here to signify that the *model* contains (in the sense of set inclusion) the *value* shown, among others. In our case, more values will be added to the model from the parsing trees of other requirements in the SRS.

Copyright © 2001 John Wiley & Sons, Ltd.

$$\text{valspace(CP FDIR state)} \supseteq \{\text{ENABLED}\}$$
$$\text{valspace(CP Lower Limit Warning state)} \supseteq \{\text{FALSE, TRUE}\}$$

In particular, the DEFVAL model is obtained directly from the two DEFAULTVAL trees** in Figure 2, while the VALSPACE model is obtained from the various comparison and assignment subtrees (e.g., RELOP, SET, etc.), selecting only those that include literal values and including the default assignment already computed by the DEFVAL model.

Event–condition–action tables are computed in a similar way, and the resulting models for our single requirement d is:

ecatab(d) = ⟨ {CP HW data less than CP Lower Limit for 3 sampling},
{CP FDIR state equals ENABLED},
{SET CP Lower Limit Warning State to TRUE;
ISSUE Warning Level Alarm "Node 1 Cabin Pressure etc."}
⟩

whose interpretation coincides largely with the original requirement, once we ignore temporal details, initial values and cross-references. If the entire specification of the Cabin Pressure Monitoring function had consisted of our sample requirement only, many validation properties (step 7) would have been violated. For example, the CP FDIR state that is mentioned in the requirement has no default value (first violation) and it only has a single possible value (second violation) since it is neither assigned to nor compared with the corresponding DISABLED value in this requirement. Indeed, such assignments and comparisons are included in other requirements in the SRS, and both the definition and the use of that particular data item in the specification are correct and consistent.

## 5. FINDINGS OF THE VALIDATION

In this section we present our actual findings from the case study. Each finding is introduced first by presenting the formal property that was violated and, where applicable, the actual model on which the property failed, then by investigating the causes of the violation and finally by suggesting possible remedial actions.

*Finding 1.* 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

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

We found six different data items (listed in Table I) that did not satisfy this simple property. Closer inspection triggered by this finding revealed that several data items whose labels started with 'ACS

**Notice that in defval(CP Lower Limit) the default value is actually represented as a pointer to the UNITVAL (value with a measuring unit) subtree of the corresponding parse tree. We use the unparsed representation here to simplify the notation. We use the same convention for the ECATAB model.

Copyright © 2001 John Wiley & Sons, Ltd. 

Table I. 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 Aalarm" | {"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 |

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. Once equivalent names had been declared as synonyms, the corresponding data items passed the validation check.

*Finding 2.* 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 (that was not referenced in the requirement) confirmed that issuing an alarm was the operation dual to setting it to normal and 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.

*Finding 3.* Another issue related to alarms was that the SRS referred to the same alarm in different ways. For example (letters refer to requirements in the SRS; we only quote the relevant fragments below):

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, the one we used as an example in Section 4) 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

Copyright © 2001 John Wiley & Sons, Ltd. 

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.

*Finding 4.* 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.

*Finding 5.* A better understanding of these potentially dangerous conditions was gained by looking at the ECATAB model. This model 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. The ECATAB model synthesized by our tool from the NL text of the requirements is shown in Table II, where we substitute abbreviations for the very long data and command names used in the SRS document. Table II uses a tabular notation inspired by the SCR notation [25]: for each requirement $r$, and for each triple in ecatab($r$), conjuncts in the *events* predicate are marked with '@T' (read: 'becomes true') or '@F' (read: 'becomes false') if they are negated; conjuncts in the *conditions* predicate are marked with 'T' (read: 'is true') or 'F' (read: 'is false') if they are negated; elements of the *actions* set are explicitly listed under the Actions heading.

For example, the row labelled d in Table II may be read as 'when the read Pressure (P) is lower than the Lower Limit (LL) for three sampling cycles, if FDIR is ENABLED, then NCS shall set the Lower Limit Warning State (LLWS) to TRUE, and set the Lower Limit Warning (LLW) to "in alarm". This row corresponds to the requirement that we used as an example in Section 4.

This table passes the usual consistency checks; e.g., disjointness and coverage, under the customary one-input assumption [26]. This assumption states that exactly one variable changes value between the executions of two sets of actions—in other words, that the system reacts to a single change at a time. As an immediate consequence, columns containing a single '@T' cannot cause any inconsistency, since in this case only one action set will be executed. The only exception in Table II is column D.FDIR CC ("Disable FDIR Confirmation Command"). However, the actions corresponding to row $i_1$ (respectively $i_2$) are taken only subject to the condition that CCREQ = Enabled is true (respectively false), so no inconsistency can arise between the two rows.

Actually, the one-input assumption holds in our case for events triggered by the reception of commands—namely, D.FDIR, D.FDIR CC, E.FDIR—due to the serial nature of the 1553 bus used to carry them. However, we do not know if the assumption 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. It is interesting to note that the VALSPACE model shows that FDIR and CCREQ can be either ENABLED or DISABLED only, so that checking for either value in Table II is sufficient. The table also shows that the system exhibits a certain hysteresis, simply maintaining the previous state for boundary cases where Pressure exactly equals the Upper or Lower Limit.

Copyright © 2001 John Wiley & Sons, Ltd.

Table II. ECATAB model.

| | Events | | | | | | | | Conditions | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Req. | Sample cycle (1 Hz) | P < LL × 3 s.c. | P > LL × 3 s.c. | P > UL × 3 s.c. | P < UL × 3 s.c. | D.FDIR | D.FDIR CC | E.FDIR | FDIR = ENABLED | CCREQ = ENABLED | Actions |
| a | @T | | | | | | | | | | Acquire P |
| b | | | | | | | | | | | |
| c | | | | | | | | | | | |
| d | | @T | | | | | | | T | | LLWS := TRUE<br>LLW := IN_AL |
| e | | | @T | | | | | | | | LLWS := FALSE<br>LLW := RTN |
| f | | | | @T | | | | | T | | ULWS := TRUE<br>ULW := IN_AL |
| g | | | | | @T | | | | | | ULWS := FALSE<br>ULW := RTN |
| h | | | | | | @T | | | | | CCREQ := ENABLED<br>CCREJ := FALSE |
| $i_1$ | | | | | | | @T | | | T | FDIR := DISABLED<br>CCREQ := DISABLED |
| $i_2$ | | | | | | | @T | | | F | Reject D.FDIR CC<br>CCREJ := TRUE |
| j | | | | | | | | @T | | | CCREQ := DISABLED<br>FDIR := ENABLED<br>LLWS := FALSE<br>ULWS := FALSE<br>LLW := RTN<br>ULW := RTN |

Copyright © 2001 John Wiley & Sons, Ltd.

Given a set of actions $A$, we define its *write-set*, $\mathrm{W}(A)$, as the set of data items modified by the actions in the set, that is

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

The intuition behind this definition is that the value of a data item $v$ is changed either by an explicit assignment of a literal value to it (in which case the new value is known), or by the special Acquire action that reads a value from a hardware sensor (in which case the new value is not known). Using Table II, we identified the conditions under which the two variables identified in Finding 4, Cabin Pressure (P) and Confirmation Command Rejection Indicator (CCREJ), are not initialized (by simply inspecting the rows $r_i$ that satisfy $\mathrm{P} \in \mathrm{W}(\mathrm{actions}(r_i))$ and $\mathrm{CCREJ} \in \mathrm{W}(\mathrm{actions}(r_i))$, 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 start-up—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 $\mathrm{i}_2$, 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 behaviour is documented and does not (currently) constitute a problem.

*Finding 6.* 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, that is

$$\begin{aligned}&\mathrm{incompatible}(A, B)\\ &\quad \Leftrightarrow \exists v \in \mathrm{W}(A) \cap \mathrm{W}(B) \text{ s.t. } (v := a \in A \ \wedge \ v := b \in B \ \wedge \ a \neq b) \ \vee \ (\text{Acquire } v \in A \cup B)\end{aligned}$$

(for our purposes, since 'Acquire $v$' assigns to $v$ an unknown value, the action is potentially incompatible with any other assignment). 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, that is

$$\begin{aligned}&\forall r_1, r_2 \in \text{Requirements}, \forall \langle evt_1, cond_1, act_1\rangle \in \mathrm{ecatab}(r_1), \forall \langle evt_2, cond_2, act_2\rangle \in \mathrm{ecatab}(r_2),\\ &\qquad \mathrm{incompatible}(act_1, act_2) \Rightarrow \neg(cond_1 \wedge cond_2 \wedge evt_1 \wedge evt_2)\end{aligned}$$

In Table II, the following rows are incompatible: (d, e), (d, j), (f, g), (f, j), (h, $\mathrm{i}_1$), (h, $\mathrm{i}_2$), (h, j), ($\mathrm{i}_1$, j), but only the first and third pairs satisfy the above property. In fact, they can be discounted by simple arithmetic. For example, let us consider the first pair. Formally, from Table II we can see that

$$\begin{aligned}&\mathrm{ecatab(d)} = \langle\{\mathrm{P} < \mathrm{LL} \times 3 \text{ s.c.}\}, \{\mathrm{FDIR} = \mathrm{Enabled}\}, \{\mathrm{LLWS} := \mathrm{TRUE};\ \mathrm{LLW} := \mathrm{IN_ALARM}\}\rangle\\ &\mathrm{ecatab(e)} = \langle\{\mathrm{P} > \mathrm{LL} \times 3 \text{ s.c.}\}, \{\}, \{\mathrm{LLWS} := \mathrm{FALSE};\ \mathrm{LLW} := \mathrm{RETURN_TO_NORMAL}\}\rangle\end{aligned}$$

The actions for these two requirements are incompatible (in fact, they assign different values to LLWS), but they are never executed simultaneously, since the conjunction of their events and conditions is always false (in our case, the conjunction represents the case when the Pressure becomes at the same time higher and lower than the admissible Lower Limit, while FDIR is enabled). The

Copyright © 2001 John Wiley & Sons, Ltd. 

second and fourth cases of incompatibility are worth flagging as potentially problematic. We have in fact

$$\mathrm{ecatab}(\mathsf{d}) = \langle\{P < LL \times 3 \text{ s.c.}\}, \{FDIR = Enabled\}, \{LLWS := TRUE;\ LLW := IN_ALARM\}\rangle$$
$$\mathrm{ecatab}(\mathsf{j}) = \langle\{\}, \{ENABLE.FDIR\}, \{FDIR := ENABLED;\ LLWS := FALSE;\ LLW = RETURN_TO_NORMAL; \ldots\}\rangle$$

(with incompatible assignments to LLWS and LLW) and

$$\mathrm{ecatab}(\mathsf{f}) = \langle\{P > UL \times 3 \text{ s.c.}\}, \{FDIR = Enabled\}, \{ULWS := TRUE;\ ULW := IN_ALARM\}\rangle$$
$$\mathrm{ecatab}(\mathsf{j}) = \langle\{\}, \{ENABLE.FDIR\}, \{FDIR := ENABLED;\ ULWS := FALSE;\ ULW := RETURN_TO_NORMAL; \ldots\}\rangle$$

(with incompatible assignments to ULWS and ULW). These cases represent the situation 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 could be avoided by checking that FDIR = Enabled is FALSE as a condition for performing the actions in requirement j. This would make the conjunction of the events and actions false in both cases, thus satisfying the property above. It is probable that the original intent of the developers in requirement j was to reset the various alarms when the FDIR feature was turned on. The problem with this requirement lies in failing to check that FDIR is currently disabled before enabling it and performing a global reset. As for other findings, the situation should be discussed with the developers in order to assess the problem and to decide on appropriate corrections.

The analysis of the remaining cases, $(\mathsf{h}, \mathsf{i}_1)$, $(\mathsf{h}, \mathsf{i}_2)$, $(\mathsf{h}, \mathsf{j})$ and $(\mathsf{i}_1, \mathsf{j})$, shows that they formalize bus protocol violations and can be ignored for a serial bus.

## 6. LESSONS LEARNED

The findings described in the previous section suggest a number of lessons we can learn from our experience.

*Lesson 1.* Style manuals are already out there. We can use them.

Many organizations have style manuals used routinely in all their projects. These manuals usually contain prescriptions of the structure of requirements documents and of the kind of language that can be used, and often come from industry-wide standards (e.g., AECMA Simplified English). Even when strict standards are not adopted, internal guidelines tend to prescribe a more precise writing style by limiting the richness of unrestricted natural language. In our case study, requirements were written in a rather technical language. This allowed us to obtain complete parsing of the entire specification of the Cabin Pressure Monitoring function with only a handful of rules, whereas full natural language parsers often have several thousand rules.

The use of style manuals makes natural language requirements more amenable to automatic processing, even when using lightweight methods. Moreover, since they are already in use, there is no training cost associated with the introduction of such methods. Indeed, in our case study, instantiating the parsing stage to the language already in use at NASA incurred a cost that was a very small fraction

Copyright © 2001 John Wiley & Sons, Ltd. 

of the cost of re-training all the analysts and engineers to a new specification language or method would have cost.

*Lesson 2.* Introducing lightweight validation can be easy.

In our experience, the total effort spent on initial set-up, done once only by one of the authors—familiar with the parsing and modelling environment but completely ignorant of the problem domain[††]—was less than three working days. Moreover, doing the validation did not require any cooperation on the part of the requirements engineers and, even if performed on a larger scale, would not have added any burden to their everyday work. We believe that lightweight validation of this kind could be introduced as part of a quality improvement effort without incurring any significant training costs and without having to overcome any resistance to innovation.

*Lesson 3.* Lightweight validation is cheap. We can do it often.

Parsing the NCS specification for the Cabin Pressure Monitoring function took less than 10 s, while generating and validating the various models took between 0.5 and 2 s 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.

This means that lightweight validation can be performed often, possibly many times during the day, even to simply test the effects of experimental changes to the requirements. For the requirements writer, having immediate feedback for such experimental changes is an important factor that simplifies the analysis of alternative formulations of the requirements.

*Lesson 4.* Lightweight validation can discover subtle errors in requirements.

Lightweight validation has sometimes been associated with the identification of trivial errors only. While even trivial errors in the requirements can introduce serious defects in the final system, and are thus worth flagging and correcting as soon as possible, we have shown that lightweight techniques can also discover subtle errors in the requirements. In fact, while many of the simpler errors (i.e. Findings 1–3) were immediately identified through human inspection by the NASA Independent Validation and Verification (IV&V) facility and corrected in the first subsequent revision of the NCS specification, others (i.e. Findings 4–6) were not identified and could potentially cause problems in the system.

In fact, 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. 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 problems 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

[††]This can be an advantage in trying to identify 'obvious' errors [27].

Copyright © 2001 John Wiley & Sons, Ltd. 

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.

## 7. RELATED WORK

The idea that requirements can be analysed in an automatic fashion, in order to identify and possibly correct several kinds of errors, has always attracted much attention. Most proposals call for a formal specification of the requirements to begin with. Jackson and Damon [28] in their validation tool Nitpick assume that requirements are expressed in a subset of Z and Reubenstein and Waters [29] call for 'natural language-like' requirements written in LISP. Heitmeyer *et al.* [25] 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. In their proposal, requirements must be expressed in a tabular notation based on finite-state machines and event/conditions tables. Validation via model checkers like Nitpick or SPIN [30] can provide a high degree of confidence, but the formalization step itself is prone to errors, so some connection to informal requirements is often sought.

Duffy *et al.* [31] 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., [32]) propose to *generate* a natural language paraphrase of a formal requirements document to help the 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 that of Rolland [8], where parsing techniques are used to extract a conceptual model from natural language requirements, and that of Macias and Pulman [9], in which a strict syntax is imposed on the requirements to ensure syntactic quality. Both works use domain-independent natural language parsing, and only provide *linguistic* models of the requirements, thus missing important opportunities for validation of the *system* described by the requirements.

Mich [11] presents NL-OOPS, a tool that supports requirements analysis by generating object-oriented models from unrestricted NL requirements. Full NL analysis is performed using the NL processing system LOLITA [33] for the parsing stage. The object-oriented analysis module implements an algorithm for the extraction of the objects and their associations from the semantic net created by the parser. While LOLITA can correctly parse texts of great complexity, the simple object-association model that is extracted from the requirements provides few opportunities for validation. A comparable model is obtained in [13], where a 'utility language' based on a restricted grammar is used to express the requirements. A restricted language, specifically devoted to expressing temporal relationships, is also used in [10]. The authors show how requirements expressed in the restricted language can be mapped to the action logic ACTL; various reachability and liveness properties are then model checked on the ACTL specification. All these works focus on a single model of the system and the issue of *adaptability* to particular domain-specific needs is not discussed. The single-model view restricts the

Copyright © 2001 John Wiley & Sons, Ltd. 

applicability of the techniques described to specific cases, whereas our approach explicitly requires a customization phase on the language, on the models and on the validation properties that are important in a certain context.

## 8. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a structured approach for validating NL 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. Moreover, the low computational and human costs strongly suggest that our techniques could be used successfully *during* requirements writing and evolution[‡‡]. We believe that the application of similar 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 [34] and to some extent validating changes that are requested and implemented.

In an industrial context, our approach has the added advantage of explicitly defining (in the set-up phase) the language to be used in requirements documents, the associated models and the validation properties to be checked, actually enforcing the use of style and quality guides in the organization because they are embedded into the tool, thus supporting improved documentation and repeatability. It becomes possible to attain constant and predictable precision of the validation during the production phase, whereas traditional human inspections can suffer from distraction, boredom or simply from changes in the inspector's effectiveness due to changing personnel. With lightweight validation, an inspector can delegate the most tedious aspects of validation to automatic tools and concentrate on more challenging issues.

Although we examined a single requirements document, the techniques we developed for the case study were re-used 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. However, to give a full assessment of the impact of lightweight formal methods based on parsing of NL requirements, a larger-scale study should be conducted along the lines of the study we presented in this work, while at the same time cooperating more actively with the requirements engineers. The NCS requirements we examined were remarkably structured and regular, and used a very precise language on a well-defined vocabulary. These properties made it possible to parse the specification completely by using a small set of parsing rules. However, more complex or looser language would not necessarily entail more complex parsing rules. Rather, the extraction of models via partial parsing can be understood as a kind of information extraction, in which only those fragments in the text that contain information relevant for the synthesis of a particular model are identified and processed. By working with partial analysis,

[‡‡]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 less than a second on average and is influenced only by the amount of change in the document between validation iterations and not by the document size. In our experience, the amount of change between revisions tends to remain constant and so does parsing time.

Copyright © 2001 John Wiley & Sons, Ltd. 

the approach we presented can thus scale to larger and more complex documents without requiring a correspondingly larger set-up effort.

The inconsistencies we identified in our case study currently appear to be non-critical; however, unfortunate experiences (such as the Ariane 5 disaster [35]) 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 NL requirements documents is therefore invaluable.

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. Given the negligible cost of the application of lightweight validation, we believe that performing *continuous* validation inside a requirements authoring environment will improve the quality of requirements submitted to more traditional inspections, thus lowering the cost of the inspections themselves.

## ACKNOWLEDGEMENTS

We would like to thank Khalid Lateef and John Hinkle of the NASA IV&V Facility, West Virginia, U.S.A., for providing us with the case study and for their valuable feedback and discussions. Thanks also go to Jack Callahan and Steve Easterbrook for many useful insights into NASA V&V procedures, and to Connie Heitmeyer and Ramesh Bharadwaj for their helpful comments on an early draft of this paper. We would also like to thank the anonymous reviewers for their many comments and suggestions. The authors acknowledge the financial support of NASA under Cooperative Agreement #NCC 2-979, the UK EPSRC for the MISE (GR/L 55964) and VOICI (GR/M 38582) projects, the Italian MURST for the AI:IA project, and the EU for the RENOIR and PROMOTER 2 projects. This paper is a revised and expanded version of [36].

## REFERENCES

1. Feather MS. Rapid application of lightweight formal methods for consistency analyses. *IEEE Transactions on Software Engineering* 1998; **24**(11):949–959.
2. Jackson D, Wing J. Lightweight formal methods. *IEEE Computer* 1996; **29**(4):21–22.
3. Easterbrook S, Lutz R, Covington R, Kelly J, Ampo Y, Hamilton D. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering* 1998; **24**(1):4–14.
4. Kemmerer RA. Integrating formal methods into the development process. *IEEE Software* 1990; **7**(5):37–50.
5. Goldin L, Berry DM. Abstfinder: A prototype natural language text abstraction finder for use in requirement elicitation. *Automated Software Engineering Journal* 1997; **4**(4):375–412.
6. Hesketh J, Robertson D, Fuchs N, Bundy A. Lightweight formalisation in support of requirements engineering. *Automated Software Engineering* 1998; **5**(2):183–210.
7. Murphy GC, Notkin D. Lightweight source model extraction. *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995. ACM Press: New York, 1995; 116–127.
8. Rolland C. A natural language approach for requirements engineering. *Advanced Information Systems Engineering* (*Lecture Notes in Computer Science*, vol. 593). Springer: Paris, 1992.
9. Macias B, Pulman SG. Natural language processing for requirements specification. *Safety-critical Systems*. Chapman and Hall: London, 1993; 57–89.
10. Fantechi A, Gnesi S, Ristori G, Carenini M, Vanocchi M, Moreschini P. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design* 1994; **4**(3):243–263.
11. Mich L. NL-OOPS: From natural language requirements to object oriented requirements using the natural language processing system LOLITA. *Journal of Natural Language Engineering* **2**(2):161–187.
12. Fuchs NE, Schwertel U, Schwitter R. Attempto controlled English—not just another logic specification language. *Proceedings of the 8th International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'98)*, Manchester, UK, June 1998 (*Lecture Notes in Computer Science*, vol. 1559). Springer: London, 1999.

Copyright © 2001 John Wiley & Sons, Ltd. 

13. Juristo N, Moreno AM, López M. How to use linguistic instruments for object-oriented analysis. *IEEE Software* 2000; **17**(3):80–89.
14. Boehm BW. Verifying and validating software requirements and design specifications. *IEEE Software* 1984; **1**(1):75–88.
15. Nuseibeh B, Easterbrook S. Requirements engineering: A roadmap. *The Future of Software Engineering, ICSE-2000*, Limerick, Ireland, 6–10 June 2000, Finkelstein A (ed.). ACM Press: New York, 2000.
16. Macias B, Pulman SG. A method for controlling the production of specifications in natural language. *The Computer Journal* 1995; **38**(4):310–318.
17. NASA/Boeing. Software requirements specification for the NCS MDM CSCI. *International Space Station Document S684-10174*, April 1997.
18. Russo A, Nuseibeh B, Kramer J. Restructuring requirements specifications: A case study. *Proceedings of 3rd IEEE International Conference on Requirements Engineering (ICRE'98)*, 51–60, Colorado Springs, CO, April 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998.
19. Russo A, Nuseibeh B, Kramer J. Restructuring requirements specifications. *IEE Proceedings: Software* 1999; **144**(1):44–53.
20. Fabbrini F, Fusani M, Gervasi V, Gnesi S, Ruggieri S. Achieving quality in natural language requirements. *Proceedings of the 11th International Software Quality Week*, San Francisco, May 1998.
21. Ambriola V, Gervasi V. Processing natural language requirements. *Proceedings of the 12th IEEE Conference on Automated Software Engineering*, November 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 36–45.
22. Gervasi V. Environment support for requirements writing and analysis. *PhD Thesis*, University of Pisa, February 2000.
23. Krogstie J, Lindland OI, Sindre G. Towards a deeper understanding of quality in requirements engineering. *Proceedings of the 7th International CAiSE Conference*, Jyvaskyla, Finland (*Lecture Notes in Computer Science*, vol. 932). Springer: Germany, 1995; 82–95.
24. Aho AV, Kernighan BW, Weinberger PJ. *The AWK Programming Language*. Addison-Wesley: Reading, MA, 1988.
25. Heitmeyer C, Kirby J, Labaw BG, Bharadwaj R. SCR*: A toolset for specifying and analyzing software requirements. *Proceedings of 10th Annual Conference on Computer-Aided Verification (CAV'98)*, Vancouver, Canada, 1998.
26. Heitmeyer C, Jeffords RD, Labaw BG. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(3):231–261.
27. Berry DM. The importance of ignorance in requirements engineering. *Journal of Systems and Software* 1995; **28**(2):179–184.
28. Jackson D, Damon CA. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering* 1996; **22**(7):484–495.
29. Reubenstein HB, Waters RC. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering* 1991; **17**(3):226–240.
30. Holzmann GJ. Proving properties of concurrent systems with SPIN (*Lecture Notes in Computer Science*, vol. 962). Springer, 1995.
31. Duffy D, MacNish C, McDermid J, Morris P. A framework for requirements analysis using automated reasoning (*Lecture Notes in Computer Science*, vol. 932). Springer: Germany, 1995.
32. Dalianis H. A method for validating a conceptual model by natural language discourse generation. *Advanced Information Systems Engineering* (*Lecture Notes in Computer Science*, vol. 593). Springer: Germany, 1992.
33. Morgan R, Garigliano R, Callaghan P, Poria S, Smith M, Uranowicz A, Collingham R, Costantino M, Cooper C, LOLITA Group. Description of the LOLITA system as used in MUC-6. *Proceedings of the 6th ARPA Message Understanding Conference*. Morgan Kaufmann, 1996.
34. Ambriola V, Gervasi V. Process metrics for requirements analysis. *Proceedings of the 7th European Workshop on Software Process Technology*, February 2000 (*Lecture Notes in Computer Science*, vol. 1780). Springer: Germany, 2000; 90–95.
35. Nuseibeh B. Ariane 5: Who Dunnit? *IEEE Software* 1997; **14**(3):15–16.
36. Gervasi V, Nuseibeh B. Lightweight validation of natural language requirements: A case study. *Proceedings of the 4th IEEE International Conference on Requirements Engineering (ICRE-2000)*, Schaumburg, IL, USA, June 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.


Copyright © 2001 John Wiley & Sons, Ltd. 