Using a SMT solver for risk analysis : detecting logical mistakes in texts

Florence Bannay[1] Marie-Christine Lagasquie-Schiex[1] William Raynaut[1] Patrick Saint-Dizier[1]
[1] IRIT, Université de Toulouse, France

Résumé

Cet article décrit une contribution de l'intelligence artificielle dans un domaine spécifique : l'analyse des risques dus à la mauvaise rédaction des documents de support technique. Ce travail combine le traitement du langage naturel et la vérification logique de satisfaisabilité. Nous expliquons comment les tests de satisfaisabilité peuvent permettre de détecter des incohérences, des redondances ou des incomplètudes dans les textes procéduraux et nous décrivons l'outil développé.

Mots Clef : Satisfaisabilité, traitement du langage naturel.

Abstract

The purpose of this paper is to describe a contribution of Artificial Intelligence to a special domain : the analysis of the risks due to poorly written technical documents. This contribution combines natural language processing with logical satisfiability checking. This paper explains how satisfiability checking can be used for detecting inconsistencies, redundancy and incompleteness in procedural texts and describes the implemented tool.

Keywords : Satisfiability, natural language handling.

1 Introduction

Companies maintain a large number of procedural texts in various sectors which may lead to risky situations. Poor requirement compliance in procedures often leads to accidents, with major health and ecological consequences. Social and psycho-social problems (*e.g.* due to poor management requirements) are also often encountered and, obviously, negative financial situations may be critical. The negative consequences of bad or poor requirements and procedures have been investigated in depth. Industry data show that approximately 50% of product defects originate from incorrect or unreadable procedures. Perhaps 80% of the rework effort on a development project can be traced to requirements defects. Because these defects are the cause of over 40% of accidents in safety-critical systems (see [8]), poor requirements and procedures have even been the ultimate cause of both death and destruction.
The LELIE [1] project was funded by the ANR Emergence [2], combining ergonomy, language processing and artificial intelligence with an applicative orientation. This project aims at detecting potential risks in industrial documents based on language processing and logic-based artificial intelligence techniques. This implies the analysis of risks indicators of different kinds (health, ecology, economy, etc.).
In this paper, we present an original appoach proposed in LELIE where natural language processing combined with AI is used for an analysis of inconsistencies, redundancies and incompleteness typical of technical documents. We concentrate on *procedural documents and requirements* (*e.g.* for installation, production, maintenance) which are, by large, the main types of technical documents.
Given a set of procedures over a certain domain produced by a company, and possibly given some domain knowledge (ontology or terminology and lexical data), the goal is to detect and model these errors and then to annotate them wherever potential risks are identified. Procedure authors could then be invited to revise these documents. Risk analysis is based on several types of considerations :

1. Inappropriate ways of writing that may lead to potential risks : texts including a large variety of complex expressions, fuzzy terms, implicit elements, scoping difficulties (connectors, conditionals), lack of cohesion, inappropriate granularity level, etc. These inappropriate ways were established by cognitive ergonomic simulations and analysis (see [1]).
2. Incoherence among procedures : detection of unusual ways of realizing action (*e.g.* unusual instrument, temperature, length of treatment, etc.) with regard to [3] similar actions in other procedures. This was based on a repository of actions from previously processed procedures (Arias software) on a given domain.
3. Lack of compliance of procedures wrt domain security requirements and regulations, therefore leading to risks. Inconsistencies or incompleteness situations are often observed between procedures and requirements.

Only Point 3 is developed in this paper (the study of Points 1 and 2 is mainly based on the way documents are written [4]). This point mainly deals with contents aspects and requires several types of inferences and reasoning.The goal of this paper is threefold. First it demonstrates the feasability of a tool that can automatically detect potential risks in natural languages technical documents. It also shows the benefits of logic-based tools like SMT-solver and

1. LELIE : An intelligent assistant for the analysis and the prevention of risks in industrial processes, project realized from 2011 to 2013 by IRIT in collaboration with CRTD-CNAM, Paris
2. ANR : Agence Nationale pour la Recherche ; a French government agency.
3. wrt for short
4. That also corresponds to a major problematics in industry, see [16].

theoretical concepts such as ATMS for industrial-oriented applications. Third it shows that it is possible to combine a natural language analysis with a logical handling of inconsistency. Moreover, we show that those tools allow us not only to detect the existence of a problem but also to point out the parts of the text that are responsible of it. The identification of major errors in procedures wrt related requirements is a very important result achieved by the LELIE tool.
Section 2 briefly explains the automatic analysis of the structure of requirements and of procedures done with the system TEXTCOOP in order to produce a translation into a logical form. Then the logical handling part is presented in Section 3 based on the outputs of the language processing step carried out with TEXTCOOP. The corresponding implemented tool is described in Section 4. Section 5 gives some related works and directions for further research.

2 Natural language analysis

Procedural texts and requirements are written in specific forms and are often very well structured, hence they are less complex, in terms of structure and ambiguity, to analyze and translate into a logical form. Indeed, procedures are often presented under the form of a list of instructions, each instruction being expressed in a simplified and standard way following guidelines.

2.1 TEXTCOOP engine

The linguistic part of LELIE, is based on the TEXTCOOP system (see [15, 16]), a system dedicated to language analysis, in particular discourse (including the taking into account of long-distance dependencies). The kernel of the system is written in Prolog SWI, with interfaces in Java. Briefly, TEXTCOOP identifies (and annotates in XML) the following structures which are of interest for our purpose :

- titles, instructions, requirement statements, prerequisites, definitions, warnings, advice and some form of explanation which are proper to technical texts,
- themes, topics, strengths (for requirements),
- the main verb and its complements, in particular instruments (with equipment or product names) or adjuncts such as amounts which are numerical values (Ph, Volts, weights, etc.) and temporal complements.

2.2 Procedural texts tagged by TEXTCOOP

The content analysis is done on three kinds of input data :

- requirements : information describing the context and the precautions with which a certain action (included into a procedure) must be carried out,
- procedures : ordered sequences of instructions,
- a list of synonyms enabling us to restrict the vocabulary and manage the term matching aspects between requirements and the related procedures.

Very simple (for readability) examples are provided below.

Example 1 *Short extract of a tagged instruction :*

```
<procedure> <predicate>use</predicate> <object>a
rope</object> to tie the harness. </procedure>
```

The verb 'use', in the instruction of this procedure, is a predicate that takes as arguments a subject (here the person who executes the procedure, called the operator, `op` *for short) and a complement/adjunct (here "the rope"). So, this sentence can be formally written as :* `use(op, rope)`. *The remainder of the instruction is ignored.*

```
<procedure> in order to <theme>sweep a chimney</theme>
<predicate>climb</predicate> <location>on the roof
</location> </procedure>
```

There are two parts in this instruction ; the first one can be translated as previously whereas the second one gives the theme of the procedure, i.e. the execution context of the procedure ; it is expressed by the formula `is(theme,sweep_a_chimney)`.

Example 2 *A set of requirements :*

```
<requirement> in case of <theme>work at a height</theme>
<predicate>do not use</predicate> <object>ropes</object>
</requirement>
```

Here the verb (with its subject and complements/adjunct) is translated into `¬use(op,rope)`. *It is given with a theme :* `is(theme,work_at_a_height)` *and they are linked :* **if** `is(theme,work_at_a_height)` **then** `¬use(op,rope)` *; (material implication).*

Example 3 *This example is a part of a synonym file :*

```
SYN = work at a height
sweep a chimney
work on roof
```

These data are used for simplifying the texts (requirements or procedures). They come from domain knowledge. By using this synonym file, the themes of Example 1 and of Example 2 become identical, enabling us to detect logical incorrectness. This method of simplification is rough (an improvement could be to use domain ontologies).

3 Logical mistakes detection

The solution proposed for answering to Point 3 of Introduction is a tool that applies basic AI reasoning principles to validate the texts written under a logical form. In our proposal, this validation is done on the three following points : inconsistency, incompleteness and redundancy detection. This is a two-step process. First, translate the written technical documents into a formal language. Then, using an open-source solver, reason on this translation.

3.1 The Logical language

We choose a representation language $\mathcal{L}$ which is a variant of a first-order logic language (see [6]). In our case, $\mathcal{L}$ is defined without symbols of function. The *terms* are classically defined using constants and variables ; *ground terms* are special terms using only constants. For any predicate symbol P of arity n, $P(t_1, \ldots, t_n)$ is an *atomic formula* (or *atom*) whenever $t_1, \ldots, t_n$ are terms. Moreover, if $t_1, \ldots, t_n$ are all ground terms then $P(t_1, \ldots, t_n)$ is a

ground atom. $\bot$ is the atomic formula representing the *contradiction*. A *literal* is an atom or its negation. Other non atomic formulas are built by using connectors, and quantifiers applied to variables and the delimitors.
The choice of this language is justified by the following facts : procedural texts are composed by simple sentences with a limited set of terms ; first-order logic is a well known language with an interesting basic expressivity ; there are many possible extensions if we want to increase this expressivity (for instance, the reintroduction of functions) ; several open-source solvers exist whose efficiency has been proved by decades of research and competitions. In this project, we choose to use the "Z3" solver (see [13]) that respects the formalism that is issued from the Satisfiability Modulo Theory (SMT) area, see [3]. The SMT library allows us to perform automated deduction and provides methods for checking the satisfiability of first-order formulas wrt some set of logical formulas T (called a theory).

3.2 Translation of the input data

Requirements and procedures are translated into first-order logic in a three-step process : 1) "clean" the text of the sentence by using the lists of synonyms and by removing the articles, and identify the theme(s), 2) find the mask corresponding to the sentence and format it wrt this mask, 3) translate the clean and formatted sentence into first-order logic, using the theme(s).
Several types of masks can be considered according to the form of the sentence :

- a simple component is a mask for a simple sentence : a verb, its subject, its complements and some adjuncts,
- a complex component is a conjunction or disjunction of simple components,
- a component with conditions and exceptions is a structure with 3 complex components.

Each simple component can be instantiated by several variants depending on the semantics of the sentence. For instance, the subject can be a constant or can be quantified universally or existentially ; the sentence can be in direct or indirect form ; the verb can be an action or not . . .
Example 1 (cont) *After cleaning (the non-tagged parts are removed and the synonyms given in Ex. 3 are replaced) :*

```
<procedure> <theme>work at a height</theme>
<predicate>climb</predicate> <location>roof</location>
<predicate>use</predicate> <object>rope</object>
</procedure>
```

Then, for each sentence, a generic mask is identified and the sentence is formatted wrt this mask. In this example, two masks are used (when an element is missing, it is replaced by NULL, except for the time-step that is encoded by an integer incremented at each instruction) :

```
theme is work_at_a_height 0
 (mask:subject state-verb attribute time)
NULL climb NULL NULL roof 1
 (mask:subject action-verb direct-obj method place time)
NULL use rope NULL NULL 2
 (mask:subject action-verb direct-obj method place time)
```

Each sentence/instruction is considered as a first-order formula that must be true at the moment corresponding to the execution of the instruction. So, this procedure corresponds to the three following first-order formulas :

```
is(theme,work_at_a_height,0)
climb(op,NULL,NULL,roof,1)
use(op,rope,NULL,NULL,2)
```

Since, the chosen solver uses the SMT formalism (see [3]), this procedure is encoded in this formalism and the resulting program code consists of :

- *first, define the different elements used in the language (here* `Agent`, `Item`, `Place`, `Attribute` *and* `Method`*) ; note that temporal elements are encoded as integers (*`Int` *is predefined in SMT) ;*
- *then, for each sentence, declare the predicate (a function in SMT), the constants, and assert the formula.*

So, the final translation of this example is :

```
(declare-sort Agent)
(declare-sort Item)
(declare-sort Place)
(declare-sort Attribute)
(declare-sort Method)
(echo "<theme> sweep a chimney</theme>")
(declare-fun is (Item Attribute Place Int) Bool)
(declare-const it_theme Item)
(declare-const att_theme_work_at_a_height Attribute)
(declare-const pl_NULL Place)
(declare-const ag_NULL Agent)
(declare-const me_NULL Method)
(assert (is it_theme att_theme_work_at_a_height pl_NULL 0))
(echo "<predicate> climb </predicate>
   <object> onto the roof </object>")
(declare-fun climb (Agent Item Method Place Int) Bool)
(declare-const it_roof Item)
(assert (climb ag_NULL it_roof me_NULL pl_NULL 1))
(echo "<predicate> use </predicate>
   <object> a rope </object> ")
(declare-const it_rope Item)
(assert (use ag_NULL it_rope me_NULL pl_NULL 2))
```

3.3 Checking correctness with a SMT solver

We propose 3 kinds of validation for procedures wrt requirements : consistency checking, incompleteness detection, and non-redundancy checking. All these validations are realized with the solver Z3 by using the notion of satisfiability of a formula (a set of formulas is handled as the logical conjunction of the formulas of the set). $\phi \models \bot$ denotes the fact that ϕ is unsatisfiable.

Notation 1 *F_i denotes the formula corresponding to the i^{th} instruction of the procedure. R denotes the formula corresponding to the set of requirements. $Lit(F) = \{l_1, ..., l_n\}$ denotes the set of (positive or negative) literals used in F.*

The detection of inconsistency can be done either on a set of requirements, or on a procedure (a set of instructions), or between a set of requirements and an instruction (or a set of instructions) :

Definition 1 (Inconsistency Detection) *Let R be a set of requirements. Let $\{F_1, \ldots, F_n\}$ be a set of instructions.*

- *There exists an inconsistency in the set of requirements iff $R \models \bot$.*
- *There exists an inconsistency in the set of instructions iff $F_1 \wedge \ldots \wedge F_n \models \bot$.*

– *There exists an inconsistency between a set of requirements and a set of instructions iff* $R \wedge F_1 \wedge \ldots \wedge F_n \models \bot$.

Example 4 *Input data are the followings :*
requirements :

```
<requirement> in case of <theme>work at a height</theme>
<predicate>be protected</predicate>
<predicate>do not use</predicate> <object>ropes
</object> </requirement>
```

instructions :

```
<procedure>in order to <theme> sweep a chimney </theme>
<predicate>climb</predicate> <location>onto the roof
</location>
<predicate>use</predicate> <object>a rope</object>
</procedure>
```

The logical translation of these data corresponds to the following formulas :

R : (`is(theme, work_at_a_height)` $\rightarrow$ `is(op, protected)`) $\wedge$(`is(theme, work_at_a_height)` $\rightarrow \neg$`use(op, rope)`)

F_1 : `is(theme, work_at_a_height)`

F_2 : `climb(op, roof)`

F_3 : `use(op, rope)`

Here, requirements are consistent ($R \not\models \bot$), instructions are consistent ($(F_1 \wedge F_2 \wedge F_3) \not\models \bot$) but requirements and instructions are inconsistent ($(R \wedge F_1 \wedge F_2 \wedge F_3) \models \bot$).

Moreover, using an ATMS (see [10]), it is possible to identify the origin of the inconsistency. This can be done very simply by first translating every formula in one or several clauses [5] then introducing a new predicate of arity 0 (called assumption predicate) *for* each clause and *in* each clause. The ATMS is able to detect the *nogoods* of the knowledge base, *i.e.* subsets N of formulas such that : 1) the formulas of N are only assumption predicates, 2) the set N is inconsistent with the knowledge base, 3) N is minimal wrt set-inclusion among the sets respecting 1) and 2).

Example 4 (cont) *In this example, the knowledge base contains 5 clauses (R produces two clauses) completed with the assumption predicates* R_1, R_2, F_1, F_2, F_3 *:*

($R_1 \wedge$ `is(theme,work_at_a_height)`) $\rightarrow$ `is(op,protected)`

($R_2 \wedge$ `is(theme,work_at_a_height)`) $\rightarrow \neg$`use(op,rope)`

$F_1 \rightarrow$ `is(theme,work_at_a_height)`

$F_2 \rightarrow$ `climb(op,roof)`

$F_3 \rightarrow$ `use(op,rope)`

Then, using an ATMS, the set $\{R_2, F_1, F_3\}$ *is a nogood that gives the origin of the inconsistency between requirements and instructions : if someone works at a height then he cannot use ropes (R_2); someone works at a height (F_1); and he uses ropes (F_3).*

Note that the computation of the nogoods is already partially implemented in the SMT-solver Z3, since it is possible to assign a name for each assertion and to extract unsatisfiable cores (*i.e.*, a subset of assertions that are mutually unsatisfiable). However this set is not guaranteed to be minimal and only one set is returned even when there are several possible causes for inconsistency [6]; so using Z3, we can directly obtain a set containing one of the nogoods. The check of non-redundancy consists in verifying that the addition of a new instruction to a set of instructions allows the inference of new formulas (otherwise it is the symptom that this new instruction is useless). This check can be formally defined as follows :

Definition 2 (Non-redundancy check) *Let* $\{F_1, \ldots, F_j\}$ *be a set of instructions. Let* F_k *be a new instruction. If* $F_1 \wedge \ldots \wedge F_j \wedge F_k \not\models \bot$ *then* F_k *is not redundant iff* $F_1 \wedge \ldots \wedge F_j \wedge \neg F_k \not\models \bot$ [7].

Example 4 (cont) *In this example, let us consider that the third instruction F_3 (which produces an inconsistency) has been replaced by the following new instruction F'_3 :*

```
<predicate>climb</predicate>
   <location>onto the roof</location>
```

This instruction is exactly the instruction F_2. So there is a redundancy that is detected as follows : $F_1 \wedge F_2 \wedge F'_3 \not\models \bot$ (no inconsistency in the procedure) and $F_1 \wedge F_2 \wedge \neg F'_3 \models \bot$ (so $F_1 \wedge F_2 \models F'_3$). This means that F'_3 is inferred by $F_1 \wedge F_2$ and so F'_3 is useless.

Note that it is possible to explain the source of redundancy (as done for inconsistency) by extracting unsatisfiable cores containing the negation of the new instruction. In the previous example we would obtain that F'_3 is redundant with $\{F_2\}$ (since $\{F_2, \neg F_3\}$ is an unsatisfiable core).
Searching for incompleteness corresponds to two distinct options that can be formally define as follows :

Definition 3 (Incompleteness detection) *Let R be a set of requirements. Let $\{F_1, \ldots, F_j\}$ be a set of instructions and F_k be a new instruction.*

– *There exists an incompleteness in the set of requirements iff there is at least a ground literal $l \in Lit(R)$ s.t. $R \models l$.*
– *There exists an incompleteness of the instruction wrt to the set of requirements iff there is at least one ground literal $l \in Lit(R \wedge F_k)$ such that $R \not\models l$, $F_1 \wedge \ldots \wedge F_j \wedge F_k \not\models l$, $R \wedge F_1 \wedge \ldots \wedge F_j \wedge F_k \models l$.*

The first point of Definition 3 is not, strictly speaking, an "incompleteness" (it rather means that the set R is too strong deductively), whereas the second point exactly corresponds to an incompleteness since it means that the union of requirements and instructions allows the inference of new formulas that are not inferred by the instructions alone (*i.e.* these instructions are too weak deductively).

Example 4 (cont) *In this example, if consider only the requirements R and the first instruction F_1 then there is an incompleteness of this instruction wrt*

5. A clause is a disjunction of atomic formulas.

6. A solution has been proposed by Liffiton and Malik [12] but it is not yet available in the standard solver.

7. That means that F_k is not inferred by $\{F_1, \ldots, F_j\}$.

requirements. Indeed, considering the ground literals that can be defined from $R \wedge F_1$, we have :

ground atom v	$\models$ *by* $R \wedge F_1$	$\models$ *by* F_1	
`is(theme, work_at_a_height)`	*Yes*	*Yes*	
`is(op, protected)`	*Yes*	*No*	$\star$
`use(op, rope)`	*No*	*No*	
`¬is(theme, work_at_a_height)`	*No*	*No*	
`¬is(op, protected)`	*No*	*No*	
`¬use(op, rope)`	*Yes*	*No*	$\star$

Using only the instruction, it is not possible to deduce the ground atoms indicated with the $\star$ symbol. This means that the instruction is incomplete wrt the requirements. Indeed, the procedure lacks at least an instruction in order to be protected and another one for forbidding the use of ropes.

4 Functional description of the tool

The logical part of the tool (described on the french website [14]) has been realized in Java ; it mainly implements four features :

1. Project definition : a project gathers several textual procedures, requirements and synonyms files, the user can create and modify projects.
2. "Cleaning" and cutting sentences : it consists in suppressing useless words and replacing some words by their standard synonyms. The tool also translates tagged sentences issued from TEXTCOOP into formatted sentences according to their mask (the tagged sentences may be requirements or instructions). At this stage "manual correction" is enabled : the user can propose other synonyms or other masks.
3. Consistency, completeness and non-redundancy checking : these functionalities are available after a translation of formatted sentences into first-order formulas using the SMT formalism. Completeness and non-redundancy cannot be performed without a previous consistency check.
4. Miscellaneous : two other functionalities have been proposed, the possibility to automatically load the requirements associated with a procedure by using its theme(s) and the possibility to take into account numeric interval values checking.

4.1 Project definition

In order to provide a convivial tool, several files can be gathered in one project. Those files and the ones that will be generated will be stored in the same directory. The interface for the project handling divides the screen in three parts, requirement files, procedure files and synonyms files. The tool enables the user to add or remove files.

4.2 "Cleaning" and cutting

This functionality consists of removing the words that are not tagged by TEXTCOOP. In a second time all articles and prepositions are removed. Moreover the text is updated in order to reduce at most the vocabulary used, this is done by using synonyms files. Before this update, the tool checks for the consistency of synonyms files in order to avoid problems like "word A should be replaced by word B", and "word A should be replaced by word C" ; then the tool does a transitive closure of the synonyms files in order to simplify cases where "word A should be replaced by word B" and "word B should be replaced by word C".
The cutting part consists in matching the clean sentence with a predefined mask. In practice this is done by studying the tags given by TEXTCOOP in order to fit the mask.
It is possible for the user to browse the different files, and to open a detailed view of these files in the main part of the screen. The different stages of the cleaning and cutting processes are shown to the user who can check for the validity of the current translation (and may alert the system if there is a wrong mask selected). Since the tool is in an experimental stage, manual corrections are available : the user is enabled to give new synonyms, to alert about some mask mistakes and to propose another tagging of some words and eventually to write an explanation/comment for the TEXTCOOP administrator.

4.3 Logical correctness

Once the clean and corrected sentences have been associated with a mask, they are translated into logic (as explained in Section 3.2), then the logical formula is sent to Z3 solver. The results are parsed in order to give a clear diagnostic and they are presented thanks to a translation into XSL which allows us to show the texts in a browser in a more convenient way, using colors and fold/unfold effects. More precisely the original sentences are shown in the order they had in the procedure, they are coloured by the system (for instance the inconsistencies appear in red while correct instructions are in green), every item is unfoldable in order to see their different translation stages and it is possible to obtain an explanation of the inconsistency by clicking on the item "inconsistency sources" (*i.e.*, nogoods).
A procedure being consistent, its completeness should be checked. An item "Completeness check" is proposed to the user and can be unfold by clicking on it. The red color is used to signal the requirements that are not fulfilled, redundancies are colored in yellow.

4.4 Miscellaneous

The tool is able to select automatically the requirements related to a procedure according to its theme. The consistency of a procedure can be checked either instruction by instruction or by checking a group of instructions together and by shifting the entire group forward of one instruction. Another feature is the ability to detect and reason about numerical values and intervals of numeric values. For instance, it is possible to use numbers in instructions or requirements "check that the sensor temperature is equal to 25", "the sensor temperature should be between 5 and 10". This has been done by adding comparison tags to the masks as well as values or interval values. In particular, it is possible to set the time as well as the place of an instruction.

The time being represented by an integer, it can be used in comparisons.

Note that due to the many possible file format for text encoding (specially for French), this encoding should be specified by the user. The tool may run on different operating systems, the operating system is detected automatically (it is necessary for a correct handling of the file storage).

5 Related works and discussion

This paper describes a part of a tool that is based on AI-techniques and automatic natural language processing. More precisely the tool is able to translate procedural texts into a predicate language in order to detect logical incorrectness. This detection is done thanks to the Z3 SMT-solver. The choice to use a SMT-solver and not a SAT-solver is justified by the fact that it is easier to translate a sentence in natural language by a logical expression using a predicative form than into an expression with propositional variables. Moreover in a SMT-solver it is possible to handle numerical values which are frequent in industrial domains, the availability of quantifiers and function symbols was also one reason for our choice even if we do not use them in the current version of the tool. The use of Prolog could also have been chosen in order to check logical inconsistencies, the benefit of SMT-solver is their efficiency in time (this is due to the SAT research progress that have been stimulated by the international competitions among SAT-solvers [9] and among SMT-solvers [2]).

Our use of the SMT-solver Z3 is a new application for this kind of solvers that were initially designed for software verification and analysis [13, 11]. It also has numerous other applications in automated theorem proving, in hardware verification [4], and in scheduling and planning problems [7] for instance. But as far as we know this is the first use of Z3 in combination with an automatic handling of natural language in order to detect logical incorrectness in texts.

The LELIE project is a new approach for analyzing texts. The existing systems were either only able to correct grammatical mistakes or only dedicated to manage the requirements files and handle requirements traceability (see [5] for a review of the existing softwares). The idea to help people to correct higher-level mistakes like logical ones is completely new in the domain of automatic and interactive correction of written texts. The checks that are carried out by our tool are crucial to reduce complexity and mistakes in industrial texts hence to prevent industrial risks.

Several directions of improvement can be considered, e.g, :

- the use of ontologies in order to exploit the hierarchical links between the manipulated objects (for instance to exploit more intelligently the synonym files).
- a user-validation of the set of existing masks on real-cases (for instance, do we need masks containing universal/existential quantifiers ?) ; and, more generally, the use of real-case is needed for validating the principles and the tool, since the scalability of the tool is very important, it is an ongoing task,
- the automatic correction of inconsistencies (possibly by giving priorities to some requirements),
- an applet version of this tool.

Références

[1] F. Barcellini, C. Albert, and P. Saint-Dizier. Risk analysis and prevention : Lelie, a tool dedicated to procedure and requirement authoring. In *Proc. of LREC*. ACL, 2012.

[2] C. Barrett, L. de Moura, and A. Stump. SMT-COMP : Satisfiability modulo theories competition. In *Computer Aided Verification*, pages 20–23. Springer, 2005.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard : Version 2.0. In A. Gupta and D. Kroening, editors, *Proc. of the 8th Intl. WS on Satisfiability Modulo Theories*, 2010.

[4] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT (BV) solver for hard industrial verification problems. In *Computer Aided Verification*, pages 547–560. Springer, 2007.

[5] C. Ebert and R. Wieringa. Requirements engineering : Solutions and trends. In A. Aurum and C. Wohlin, editors, *Engineering and Managing Software Requirements*, pages 453–476. Springer Berlin Heidelberg, 2005.

[6] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, 1996.

[7] P. Gregory, D. Long, M. Fox, and J.C. Beck. Planning modulo theories : Extending the planning paradigm. In *ICAPS*, 2012.

[8] Health and Safety Executive. Fatal injury statistics. http ://www.hse.gov.uk/statistics/fatals.htm.

[9] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1) :89–92, 2012.

[10] J. De Kleer. An assumption-based TMS. *Artificial Intelligence*, 28 :127–162, 1986.

[11] S. Lahiri and S. Qadeer. Back to the future : revisiting precise program verification using SMT solvers. *ACM SIGPLAN Notices*, 43(1) :171–182, 2008.

[12] M. Liffiton and A. Malik. Enumerating Infeasibility : Finding Multiple MUSes Quickly. In *Proc. of CPAIOR*, pages 160–175, 2013.

[13] L. Moura and N. Bjørner. Z3 : An efficient SMT solver. In C.R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] W. Raynaut. *Module IA de l'Outil Lelie. Un logiciel intelligent d'aide au diagnostic de risques dans les procédures industrielles*. IRIT, www.irit.fr/∼Marie-Christine.Lagasquie-Schiex/Lelie, 2013.

[15] P. Saint-Dizier. Processing natural language arguments with the textcoop platform. *Argumentation and Computation*, 3(1) :49–82, 2012.

[16] P. Saint-Dizier. *Challenges of Discourse Processing : the case of technical texts*. Cambridge University Press, 2014.