JUSTIFYING SOFTWARE SYSTEMS SAFETY USING ARGUMENTS

LICENSE THESIS

Graduate: Nicoleta Catalina MARC
Supervisor: Lect.dr.eng Adrian GROZA

2014
JUSTIFYING SOFTWARE SYSTEMS SAFETY USING ARGUMENTS

1. Project proposal: In safety-critical applications it is necessary to justify the software conformance with specifications and standards. Certification bodies require the construction of assurance cases, that contain claims supported by evidence obtained during development and testing of the system. It is important to build well-structured and coherent safety cases, because wrong construction and reasoning in safety arguments can undermine system’s safety claims and lead to failures. We developed a tool that facilitate the construction and assessment of safety cases. The tool supports the Goal Structuring Notation standard for creation of safety arguments. The GSN diagrams are translated in description logic, in order to formally check various properties of the safety case.


3. Place of documentation: Technical University of Cluj-Napoca, Computer Science Department
4. **Consultants:** Lect.dr.eng. Adrian Groza

5. **Date of issue of the proposal:** November 1, 2014

6. **Date of delivery:** July 3, 2014

Graduate: Nicoleta Catalina Marc

Supervisor: Lect.dr.eng. Adrian GROZA
Declarare pe proprie răspundere privind autenticitatea lucrării de licență

Subsemnata Marc Nicoleta Catalina, legitimat(ă) cu seria AX nr. 377045 CNP 2911205012661, autorul lucrării Justifying software systems safety using arguments elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea Calculatoare în limba Engleza din cadrul Universității Tehnice din Cluj-Napoca, sesiunea Iulie a anului universitar 2013-2014, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, anularea examenului de licență.

Data Nume, Prenume

__________________________

Semnătura
# List of Tables

2.1 Problem Statement and Solution ................................................. 18

4.1 Syntax and semantics of \textit{ALC}. ............................................. 34

4.2 Retrieving information about the safety case. ............................ 39

6.1 Retrieving some information about the safety case represented in figure 6.1. 54

6.2 Retrieving some information about the safety case represented in figure 6.4. 57

6.3 Retrieving some information about the safety case represented in figure 6.6. 59

6.4 Tools Comparison Table .............................................................. 62
## List of Figures

2.1 System usecase .................................................. 20

3.1 Acedit tool screenshot .......................................... 22
3.2 AdvoCATE tool screenshot ..................................... 23
3.3 D-case editor tool screenshot .................................. 24

4.1 Elements of a safety case ...................................... 29
4.2 Goal Structuring Notation elements ............................ 30
4.3 Partial goals structure .......................................... 30
4.4 Goal structure for the process based argument. .............. 31
4.5 Goal structure for the product based argument. .............. 32
4.6 Example of safety case represented using GSN. .............. 33
4.7 System Architecture ........................................... 37
4.8 Flow Chart Querying the diagram .............................. 41
4.9 Flow Chart for validating the diagram ......................... 42

5.1 System Architecture ........................................... 43
5.2 Component Diagram ........................................... 44
5.3 Component Diagram ........................................... 45
5.4 Class diagram for console plug-in. ............................. 47
5.5 Class diagram for ontology actions plug-in. ................... 48
5.6 Class diagram for parser plug-in. ............................... 49
5.7 Class diagram for ontology plug-in. ........................... 50
5.8 Application Interface ........................................... 52

6.1 Autonomous vehicle scenario. .................................. 54
6.2 Translating the GSN diagram in a description logic Abox. .... 55
6.3 Translating the GSN diagram in Abox using our tool ........... 56
6.4 Therac 25 safety case. ........................................... 58
6.5 Translating the Therac 25 safety case in a description logic Abox. 59
6.6 Airspace-system safety case. .................................... 60
6.7 Airspace-system safety case translated in Abox using our tool 61
7.1 Autonomous vehicle scenario. .................................................. 65
7.2 Command Console. ................................................................. 66
Contents

List of Tables 9

List of Figures 11

Chapter 1 Introduction 16
  1.1 Organization of the paper 17

Chapter 2 Project Objectives and Specifications 18
  2.1 Problem Statement and Solution 18
  2.2 Goals and Objectives 18
    2.2.1 Business Goals and Objectives 18
    2.2.2 Project Goals and Objectives 19
  2.3 Functional requirements 19
  2.4 Non-functional requirements 20

Chapter 3 Bibliographic research 22

Chapter 4 Analysis and Theoretical Foundation 28
  4.1 Safety Case 28
    4.1.1 ISO26262 standard 29
  4.2 Goal Structuring Notification 31
    4.2.1 Elements of GSN 31
  4.3 Description Language 34
  4.4 Modeling the GSN Standard in Description Logic 35
  4.5 Functionality 37
    4.5.1 Diagram translation and Reasoning in safety arguments 37
    4.5.2 Validating the Safety Case 38
    4.5.3 Generation of Safety Case Metrics 40
    4.5.4 Generating Natural Language Reports on the Safety Case 40

Chapter 5 Detailed Design and Implementation 43
  5.1 System Architecture 43
    5.1.1 GSN editor 44
Chapter 6 Testing and Validation
   6.1 Testing the tool ................................................. 53

Chapter 7 User’s manual
   7.1 System Installation ........................................... 63
   7.2 User’s Manual .................................................. 64

Chapter 8 Conclusions
   8.1 Contributions and Achievements .............................. 67
   8.2 Further Work .................................................... 68

Bibliography .......................................................... 69

Appendix A Published papers ........................................ 71
Chapter 1

Introduction

Chapter 1 describes the problem statement and its solution.

In the past few years the number of safety critical systems has grown, the question is how do we know if they are safely-made and secure, any system that presents a certain level of risks must prove that its behavior should be trusted. Nowadays software safety assurance is often demonstrated by compliance with national or international safety standards. The assurance cases are used to decide if the system is safe and secure solely from the provided evidence. Their usage has grown in the last years and in certain domains, like the nuclear field, it is mandatory to build a safety case. This proves that, currently, assurance cases are the answer to our question. Assurance cases have evolved from the concept of safety cases and they contain claims supported by the evidence obtained during development and testing of the system. It is very important to build well-structured and coherent safety cases because wrong construction and reasoning in safety arguments can undermine a systems safety claims and lead to failures of the system. Over the last years, many documents have appeared about how to describe and graphically structure an assurance case. One of these documents is the Goal Structuring Notation, also known as GSN. The Goal Structuring Notation is an argumentation notation used to structure and graphically represent a safety argument. Even though we have a solid standard for representing safety cases, it doesn’t reduce the risk of building incomplete and bad-structured assurance cases because of bad assessment of the safety case, i.e the number of claims could be very high and therefore will be hard to follow by the safety engineer. A solution to this problem is to develop a tool that facilitates the construction and assessment of safety cases. The main feature of the tool is to translate the GSN graphical notation into description logic in order check the GSN model for consistency. Hence, the GSN model for a safety critical application can be specified both in graphical notation and in description logic. The advantage is that the specific reasoning services of description logic are enacted to verify the compliance of the case with the GSN standard and also to signal possible argumentation flaws.
1.1 Organization of the paper

Chapted I presents an introduction in arguing software safety domain.
In Chapter II we introduce the theme and the objectives of our system.
In Chapted III are presented all the references for the project within the project domain and related work.

Chapter IV presents the elements of a safety argument, introduces the technical instrumentation used: the Goal Structuring Notation plus description logic.
Chapter V details the architecture of the system and implementation.
In Chapter VI are presented the needed resources and steps for installing the application, and how to use it.

Chapter VII focuses on methods for testing and validating our tool.
Finally, in Chapter VIII we present the conclusion and further work.
Chapter 2

Project Objectives and Specifications

2.1 Problem Statement and Solution

For many systems that present a certain level of risk safety is a good practice to justify, prior to deployment, if and why the software behavior should be trusted. Because structured and evidence-based arguments are more and more used to describe the assurance of the system, many international standard adopted this strategy.

As explained in table 2.1, the problem arises in assessing complex safety cases and proving that the system is sufficiently secure through the sufficient solutions and evidence are provided for each stated claim in the safety case. The found solution is using ontology reasoning services because the ontology provides a source of well-defined terms that can be used in descriptions of safety case nodes. The safety case knowledge can be represented as an ontology and after queries can be run on that ontology.

| The problem of building incomplete safety cases because of bad safety case assessment | affects system’s safety and the safety engineer |
| the impact of which is incomplete description of systems assurance | a successful solution would be improve the assessment of the tool using reasoning services |

2.2 Goals and Objectives

2.2.1 Business Goals and Objectives

Nowadays many international standards require for systems that present a certain level of risk to justify, prior to deployment, why software behavior is to be trusted. Because
of this, the business goals and objectives for this project will focus on implementing a tool that:

- improves assessment of safety cases
- reduces the risks of not building well-structured and complete assurance cases
- is easy to use
- eases the work of the safety engineer
- enables the evaluation of the realism of the cases

2.2.2 Project Goals and Objectives

The project main goal is to develop a system that:

- accomplish project business goals and objectives
- supports automated construction and assessment of safety cases
- that can be used to justify the correctness of the user's critical systems and if systems obeys the international standards
- translates the safety case into description logic
- uses reasoning services for better assessment of safety cases
- validates the safety case
- generates documentation and reports for the safety case

2.3 Functional requirements

In the use case diagram from figure 2.1 we have captured the requirements of the system. Using our tool, the user should be able to:

1. build and edit safety case diagram
2. export any diagram as image (jpg or png)
3. get a file containing the translated the diagram into A-box
4. load or set the current Abox for querying
5. query the diagram from a special console used only for this
6. validate his safety case and get a file with the status of the validation

7. Generate documentation and reports

2.4 Non-functional requirements

Usability:
The user interface shall be very easy to use and intuitive. Two goals should be strived for one click away functionality intuitive interface - zero training. Also the system should prevent the user before making errors and if the errors are made the the user will we notified about the errors.

Extensibility:
Take into consideration further work, extending and adding product features.

Documentation:
An Administration Guide and a User Guide should be developed in .pdf format.
Operating constraints:
Reasoning engine RacerPro are needed in querying the safety case, plus java jRacer library for creating a java racer client connection to the engine. NaturalOwl engine is used to generate documentation of description logic in natural language.

Reusability:
Ontology code should be in a separate plug-in so that it can be reused. Do the same for the command console code.
Chapter 3

Bibliographic research

In order to build the tool we have studied several tools used for building safety cases, among them ACedit [1] and AdvoCATE [2]. Both tools are used to create safety cases and use the GSN for structuring the safety arguments.

ACedit [1] is an open-source editor used to create Assurance Cases based on the Goal Structuring Notation standard and the OMG Argumentation Metamodel. The tool can be used only for creating and editing a safety case, it lacks at the assessment of the safety-case. Our tool is an improvement of this tool, being added new features like the option of querying the diagram using ontologies, better validation, creation of reports and documentation, usage of safety metrics for the assessment of the safety cases. Figure 3.1 represents a screenshot of this tool.

![Figure 3.1: Acedit tool screenshot](default.png)

AdvoCATE [2] is an Assurance Case Automation ToolsEt, to support the automated construction and assessment of safety cases. The tool is more complex than ACedit.
AdvoCATE is an assurance case automation toolset, and has been built to support the automated construction and assessment of safety cases. The main features of the tool are: i) create and edit of assurance cases; ii) import and export of a variety of safety case formats currently those produced from the ASCE, CertWare, and D-Case tools; iii) assemble automatically fragments of safety cases; iv) Computation of metrics, direct measurement and evaluation of the safety case.

Figure 3.2 represents a screenshot of this tool.

The novelty of our approach is that the assurance case is automatically translated in description logic. The advantage is that the specific reasoning services of description logic are enacted to verify the compliance of the case with the GSN standard and also to signal possible argumentation flaws. The difference between our tool and this one is the fact that using ours the user will be able to build simple or complex queries for interrogating the diagram at any time during the development and not being limited only to metrics, this is a plus at the assessment of the diagram. D-case Editor is another tool for constructing safety cases, developed by DEOS (Dependable Embedded Operating System for Practical Uses). Similar to our tool, it is an Eclipse plugin based on the Eclipse GMF framework and supports GSN standard. It is a prototype of a dependability case editor and has a
GSN pattern library function and prototype checking function. Figure 3.3 represents a screenshot of this tool.

![Figure 3.3: D-case editor tool screenshot](image)

A tool for integrating informal and formal reasoning has been proposed in [3]. The focus is on tracing requirements from natural language representation towards formal properties verified using model checking. The tool is a plugin for Eclipse developed on top of ProR plugin for tracing natural language requirements and Rodin for modelling properties in the Event-B language.

For implementing the reasoning feature of our system we studied about description logics from the book [4]. It presents description logic plus modelling and technical details, the syntax of knowledge bases of the description logic \( \mathcal{SROIQ} \), presents the \( \mathcal{ALC} \) description logic.

In paper [5] is presented the RacerPro system, a knowledge representation system based that implements a highly optimized tableau calculus for a very expressive description logic. Advantages of using this engine is that it offers reasoning services for multiple T-boxes and for multiple A-boxes as well and it provides us an API with Racer Client for Java.

For safety cases building and assessment we studied the following articles: [6] and [7]. In the paper [6] is presented a new way to structure assurance cases using assured safety arguments. According to the authors any safety arguments has two components:
• a safety argument that documents the arguments and the evidence used to establish direct claims of system safety

• a confidence argument that justifies the sufficiency of confidence in this safety argument.

This decomposition gives greater clarity of purpose and helps avoiding the introduction of unnecessary arguments and evidence. Many difficulties are encountered when having a single argument elements that documents both direct arguments of risk mitigation and supporting arguments, for example:

• Because there is too much information in just one argument, the arguments will become too large and unwieldy, leading to difficulty in reviewing them because of the size and lack of focus.

• It is more difficult to see the incompleteness or poor structure in the safety argument

• Arguments tend to be indirect and unfocused, and the link between elements of the argument and risk is often lost.

• Arguments become difficult to build, and weaknesses of the argument are sometimes not evident and so are easily overlooked.

These difficulties are serious since they all detract from the basic purposes of using safety cases. Linking the two arguments provides a mechanism for guiding analysis of the interrelationship between safety and confidence. Both papers [6] and [7] claim that any safety argument should focus on identification, management and mitigation of hazards associated with the system. All the elements of a safety case are part of the causal chain of hazard. The safety case is based on claims, any claim is broken into sub-claims until is reached a point when the claims can be proved by the development or assessment of an artefact as evidence. Every claim and strategy adopted to support the claim should be very clearly formulated and state the context in which the argument is made. The solution proposed by the authors for this is to represent graphically the safety argument because it is clearer than through narrative text because in a narrative text is more difficult for the reader to identify the individual elements and structure of the argument. In paper [8] the authors propose a method for assessing software safety and security standards by capturing and criticising their arguments. This method assumes following three steps: argument capture, argument criticism, and issue sentencing. In the first step, argument capture, the standard requirements are identified through analysis and also the specific claims assumed by those requirements and the evidence supporting the claims. The standard’s text should be used in the captured argument as much as practical. Throughout this process, the standard should be captured as accurately as possible. The second step called argument criticism, focuses on reviewing the fragments of argument by following the next phases:

1. Take into consideration misinterpretations of the argument
2. Try to draw out implicit or explicit assumptions
3. Judge the necessity of each assumption
4. Search for errors the argument
5. Identifies where “independent” lines of reasoning depend upon common sub-arguments
6. Take into account strengthening the argument
7. Determines whether negative experience with similar systems might provide counter-evidence
8. Judges the strength of the argument

In the last step, issue sentencing, the analyst should re-examines each identified issue to determine what hazards from the standard are reflected. More about safety argument strategies have been studied from paper [9]. In his papers he investigates the issues of autonomy for safety-critical systems, the ways of safety assurance and the structure of safety arguments. He proposes two approaches for autonomous vehicle safety. The first one is the classical one uses hazard analysis approach and is based on safety barriers. This solution aims to identify event sequences leading to accidents and ways to control risks, the safety is perceived in a binary way. The second solution is based on the dynamic risk assessment approach, i.e. design a system which is able to perceive and interpret risk factors and evaluate if forwarding will lead to a safe state or ends with an accident. Autonomous systems raise problems for safety analysis and safety assurance, and therefore for certification. ISO 26262 standard and safety cases of systems obeying this standard have been studied in papers [10] and [11]. The standard for vehicles requires building a safety case for electrical/electronic that presents a certain level of risks in order to prove that the system requirements are complete and satisfied by evidence.

In the paper [12] the user presents the case study of Therac 25 machine that had massively overdosed six people. Therac 25 is a computer controlled therapy machine that can treat the patient with relatively low energy electron beams or with X-ray. From the safety analysis of the system, it can be seen that the developers focus more on the technology from older versions of this machine than on the changes introduced by the new machine, they didn’t go through all phases of the project development, residual software errors have been omitted from the analysis, many hardware safety stops were removed because they thought that the software was better. Another bad practice committed by them: not performing risk assessment for the software, reuse of older machine’s software without checking for faults and making the system hard to use by the others (the error messages were very difficult to understand because they didn’t specified the exact cause, they were like “MALFUNCTION 47” OR “VILT” and therefore the errors were ignored by the practitioners). The case presented by the author is just a start because it doesn’t
contain very detailed the system requirements, hazard identification and a risk analysis haven’t been performed, but a top level safety case can be build. The most important requirements established by the author are: more detailed error messages, the maximum limit for MeV electron treatments should be set, occurrence of errors should stop the machine, the computer should select the correct energy and modes.
Chapter 4

Analysis and Theoretical Foundation

4.1 Safety Case

Safety cases are used in risk safety domain, in case of systems that presents a certain level of risks in order to prove that the system requirements are complete and satisfied by evidence.

In this thesis the safety case is defined in the following terms:

“A safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context “

Assurance cases are build using safety arguments, each safety arguments should focus on identifying and reducing the hazards associated with the system. The safety argument is based on claims, any claim is broken into sub-claims until is reached a point when the claims can be proved by the development or assessment of an artefact as evidence. Every claim and strategy adopted to support the claim shoulf be very clearly formulated and state the context in which the argument is made so that the graphical representation of the safety case will be well structured.

The elements of any safety case are represented in figure 4.1, they are:

- Claim: the statement or assertion
- Evidence: represents the provided evidence, facts assumptions or subclaims to support the claim
- Argument: it is the linkage between the claim and the evidence
- Inference: the mechanism providing the transformational rules for the argument

The most important aspects of an safety case are:

- argument: any safety case builds an argument based on the provided evidence, according to this argument someone can reasonably determine that the system is relatively safe
4.1. SAFETY CASE

Figure 4.1: Elements of a safety case

- clear: any safety case must be clear because all the information provided must be understood by inexperienced people.
- system: a safety case can be build for any system
- acceptably: it is very hard to prove that a system is 100% safety, the goal of an assurance cases goal is to convince the others that the system is safety enough to be used.
- context: The context of using the system must be specified in the safety case, because context-free safety is impossible to argue.

4.1.1 ISO26262 standard

Many safety standards require now the construction of safety cases for systems that may present a certain level of risk.

The ISO 26262 standard states that any electrical/electronic product must ensure an acceptable level of safety and requires building a safety case, but it does not tell you the steps of building it [13]. Fig. 4.3 shows how such an analysis is performed in order to comply to the ISO26262 requirements, according to [14]. The figure presents only the “hazard analysis and risk assessment” component. The top level goal Goal1 is to show if the product ensures a sufficient and acceptable level of safety.

The user should structure the safety case into product assurance cases and process related assurance cases.

In figures 4.3, 4.4, 4.5 is developed only the “hazard analysis and risk assessment” claim of the product and is shown the corresponding process-based (Goal 2) and product-based(Goal 6) arguments.

The process-based goal Goal2 in refined in Fig. 4.4. The goal claims that the process adopted to develop the product is correct and successfully completed. Goal2 is divided, taking in account the roles (Strategy1) and activity steps(Strategy2), in 3 sub-goals:
CHAPTER 4. ANALYSIS AND THEORETICAL FOUNDATION

Goal3, Goal4 and Goal5. Goal4 claims that the hazards regarding the adapted process of building the product have been identified and classified, using the Hazard identification and analysis using HAZOP technique (HAZard and Operability analysis) to provide the evidence, representing Evidence2 node, while Goal5 claims that all the hazard have been carefully analyzed backward and forward, providing as solution hazard identification and analysis using HAZOP technique (HAZard and Operability analysis) represented as Evidence3 and Failure Modes and Effects Analysis (FMEA) procedure and Fault Tree Analysis technique (FTA) as Evidence4.

The product-based goal Goal6 is justified in Fig 4.5. This claims that the system has the required safe behavior, if something fails then the system should be able to fail in a safe way. The goal is divided in two goals: Goal7 and Goal8. Goal7 claims that all the hazards regarding the product have been found, while Goal8 states that the the effects and causes of hazardous events have been analyzed. The goals have as solution techniques
4.2. GOAL STRUCTURING NOTIFICATION

Our tool uses Goal STRucturing Notation to represent assurance cases structure, mainly because in last years, GSN has been more and more used in risk-based safety domain. GSN presents a new argumentation notation for structuring and representing graphically safety arguments.

4.2.1 Elements of GSN

Having a well defined structured, GSN standard increases the chance of identifying gaps in proving the goal and providing evidence.

According to GSN standard, the main elements of an safety argument are: goal, strategy, solution, context, assumption, justification. The Goal represents the statement that needs to be proved, it can be divided in sub-goals until its reached the level where the
 CHAPTER 4. ANALYSIS AND THEORETICAL FOUNDATION

Figure 4.5: Goal structure for the product based argument.

sub-goal can be supported by evidence. Strategy element is used to describe the method
approached by the system to prove the claim.

A Solution node provides the evidence or references to the evidence supporting the
goal. Context element provides information relevant for the corresponding goal, while the
assumption element will provide statements that are already true. Justification elements
is used to explain why the provided evidence is enough to prove the goal. These elements
can be related to each other using two relations: inContextOf , only between goal and
context, and solvedBy. Figure 4.2 contains elements of the GSN standard represented as
follows: goals with rectangular, strategies with paralelograms, evidence and solutions are
represented by circle, assumptions and justifications with ellipse, context by a rectangular
with rounded corners.

The supportedBy relation is rendered as an arrow with the solid and allows inferen-
tiol or evidential relationships to be documented. It is used to establish the following con-
nexions: goal-to-goal, goal-to-strategy, goal-to-solution, strategy-to-goal, goal-to-solution,
goal-to-justification. The inContextOf relation is represented by an arrow with empty head
and is used to declare a contextual relationship. This relation permits connection between
goal or strategies with context, justification or assumption elements.

In figure 4.6 we have an example of goal structure composed with GSN elements.
It can be seen that we have a main goal, the top level goal, which is solved by other goals,
the support goals, and strategies. Each goal, in turn, can happen in different contexts or
different assumptions about the goal can be made, and the goal could be divided in ther
sub-goals, strategies or could be supported by evidence.
Figure 4.6: Example of safety case represented using GSN.
4.3 Description Language

DLs are based on a vocabulary, also called signature, containing individual names, concept names (unary predicates) and role names (binary predicates).

In the description logic $\mathcal{ALC}$, concepts are built using the set of constructors formed by negation, conjunction, disjunction, value restriction, and existential restriction [4], as shown in table 4.1. Here, $C$ and $D$ represent concept descriptions, while $r$ is a role name. The semantics are defined based on an interpretation $I = (\Delta^I, \cdot^I)$, where the domain $\Delta^I$ of $I$ contains a non-empty set of individuals, and the interpretation function $\cdot^I$ maps each concept name $C$ to a set of individuals $C^I \subseteq \Delta^I$ and each role $r$ to a binary relation $r^I \subseteq \Delta^I \times \Delta^I$. The last column of table 4.1 shows the extension of $\cdot^I$ for non-atomic concepts.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>$\neg C$</td>
<td>$\Delta^I \setminus C^I$</td>
</tr>
<tr>
<td>conjunction</td>
<td>$C \sqcap D$</td>
<td>$C^I \cap D^I$</td>
</tr>
<tr>
<td>disjunction</td>
<td>$C \sqcup D$</td>
<td>$C^I \cup D^I$</td>
</tr>
<tr>
<td>existential restriction</td>
<td>$\exists r.C$</td>
<td>${ x \in \Delta^I</td>
</tr>
<tr>
<td>value restriction</td>
<td>$\forall r.C$</td>
<td>${ x \in \Delta^I</td>
</tr>
<tr>
<td>individual assertion</td>
<td>$a : C$</td>
<td>${ a } \in C^I$</td>
</tr>
<tr>
<td>role assertion</td>
<td>$(a, b) : r$</td>
<td>$(a, b) \in r^I$</td>
</tr>
</tbody>
</table>

Table 4.1: Syntax and semantics of $\mathcal{ALC}$.

An ontology consists of terminologies (or TBoxes) and assertions (or ABoxes).

**Definiție.** A terminology $TBox$ is a finite set of terminological axioms of the forms $C \equiv D$ or $C \sqsubseteq D$.

**Definiție.** An assertional box $ABox$ is a finite set of concept assertions $a : C$ or role assertions $(a, b) : r$, where $C$ designates a concept, $r$ a role, and $a$ and $b$ are two individuals. Usually, the unique name assumption holds within the same $ABox$.

A concept $C$ is satisfied if there exists an interpretation $I$ such that $C^I \neq \emptyset$. The concept $D$ subsumes the concept $C$, represented by $C \sqsubseteq D$, if $C^I \subseteq D^I$ for all interpretations $I$. Constraints on concepts (i.e. disjoint) or on roles (domain, range of a role, inverse roles, transitive properties), number constraints (max, min), role inheritance (parent role) can be specified in more expressive description logics$^1$.

$^1$This paper provides only some basic terminologies of description logics to make it self-contained. For a detailed explanation about families of description logics, the reader is referred to [4].
4.4 Modeling the GSN Standard in Description Logic

The relationship supportedBy, allows inferential or evidential relationships to be documented. The allowed connections for the supportedBy relationship are: goal-to-goal, goal-to-strategy, goal-to-solution, strategy to goal. Axiom A_1 specifies the range for the role supportedBy:

\[(A_1) \quad \top \sqsubseteq \forall \text{supportedBy. (Goal} \sqcup \text{Strategy} \sqcup \text{Solution)}\]

Axiom A_2 specifies the domain of the role supportedBy, axiom A_3 introduces the inverse role supports, and A_4 constrains the role supportedBy to be transitive.

\[(A_2) \quad \exists \text{supportedBy.} \top \sqsubseteq \text{Goal} \sqcup \text{Strategy}\]
\[(A_3) \quad \text{supportedBy} \equiv \text{supports}\]
\[(A_4) \quad \text{supportedBy} \sqsubseteq \text{supportedBy}\]

(define-primitive-role supportedBy :domain (or Goal Strategy) :range (or Goal Strategy Solution) :inverse supports :transitive t)

Inferential relationships declare that there is an inference between goals in the argument. Evidential relationships specify the link between a goal and the evidence used to support it. Axioms A_5 and A_8 specify the range of the roles hasInference, respectively hasEvidence, while A_6 and A_9 the domain of the same roles. Definitions A_7 and A_10 say that the supportedBy is the parent role of both hasInference and hasEvidence, thus inheriting its constraints.

\[(A_5) \quad \top \sqsubseteq \forall \text{hasInference. Goal}\]
\[(A_8) \quad \top \sqsubseteq \forall \text{hasEvidence. Goal}\]
\[(A_6) \quad \exists \text{hasInference.} \top \sqsubseteq \text{Goal}\]
\[(A_9) \quad \exists \text{hasEvidence.} \top \sqsubseteq \text{Goal}\]
\[(A_7) \quad \text{hasInference} \sqsubseteq \text{supportedBy}\]
\[(A_10) \quad \text{hasEvidence} \sqsubseteq \text{supportedBy}\]

(define-primitive-role has-inference :parent supportedBy :domain Goal :range Goal)

(define-primitive-role has-evidence :parent supportedBy :domain Goal :range (or Evidence solution))

Goals and sub-goals are propositions that we wish to be true that can be quantified as quantified or qualitative, provable or uncertainty.

\[(A_{11}) \quad \text{QuantitativeGoal} \sqsubseteq \text{Goal}\]
\[(A_{13}) \quad \text{ProvableGoal} \sqsubseteq \text{Goal}\]
\[(A_{12}) \quad \text{QualitativeGoal} \sqsubseteq \text{Goal}\]
\[(A_{14}) \quad \text{UncertaintyGoal} \sqsubseteq \text{Goal}\]
A sub-goal supports other high level goals. Each safety case has a top level Goal, which does not support other goals.

\((A_{15})\) \(\text{SupportGoal} \equiv \text{Goal} \sqcap \exists \text{supports}. \top\)

\((A_{16})\) \(\text{TopLevelGoal} \equiv \text{Goal} \sqcap \neg \text{SupportGoal}\)

(equivalent SupportGoal (and Goal (some supports *top*)))
(equivalent TopLevelGoal (and Goal (not SupportGoal)))

For each safety argument, the elements is instantiated and a textual description is attached to that individual by enacting the attribute \(\text{hasText}\) with domain \(\text{Statement}\) and range \(\text{String}\):

\((A_{17})\) \(\top \sqsubseteq \forall \text{hasText.String}\)

\((A_{18})\) \(\exists \text{hasText.Statement} \sqsubseteq \top\)

Three individuals \(gt\), \(gp\), and \(gu\) of type goal and their textual descriptions are instantiated by assertions \(f_1\) to \(f_6\):

\((f_1)\) \(gt : \text{TopLevelGoal}\)

\((f_2)\) \(gt,"\text{The system meets its requirements}" : \text{hasText}\)

\((f_3)\) \(gp : \text{ProvableGoal}\)

\((f_4)\) \(gp,"\text{Quick release are used}" : \text{hasText}\)

\((f_5)\) \(gu : \text{UncertaintyGoal}\)

\((f_6)\) \(gu,"\text{The item has a reliability of 95}\%" : \text{hasText}\)

Intermediate explanatory steps between goals and the evidence include statements, references, justifications and assumptions:

\((A_{20})\) \(\text{Explanation} \sqsubseteq \text{Statement} \sqcup \text{Reference} \sqcup \text{Justification} \sqcup \text{Assumption}\)

where these top level concepts are disjoint:

\((A_{21})\) \(\text{Statement} \equiv \neg \text{Reference}\)

\((A_{22})\) \(\text{Statement} \equiv \neg \text{Justification}\)

\((A_{23})\) \(\text{Statement} \equiv \neg \text{Assumption}\)

\((A_{24})\) \(\text{Reference} \equiv \neg \text{Justification}\)

\((A_{25})\) \(\text{Reference} \equiv \neg \text{Assumption}\)

\((A_{26})\) \(\text{Justification} \equiv \neg \text{Assumption}\)

The evidences or solutions form the foundation of the argument and will typically include specific analysis or test results that provide evidence of an attribute of the system. In our approach, the evidence consists in model checking the verification for a specification of the system.

A not verified goal is a goal which has at least one piece of evidence that is not formally proved.

\((A_{27})\) \(\text{NotVerifiedGoal} \equiv \text{Goal} \sqcap \exists \text{hasEvidence.}\)

\(\text{NotVerifiedEvidence}\)
4.5 Functionality

4.5.1 Diagram translation and Reasoning in safety arguments

The main idea of translating the built safety case diagram into A-box is presented in figure 4.7.

Figure 4.7: System Architecture

We translate the GSN standard into description language, representing the Tbox and for each diagram built the user can transform the diagram into description language, representing the A-box part. Both parts are loaded in the RacerPro reasoning engine which will be used by our tool to query and validate the Abox against the GSN Tbox.

Querying the Diagram Scenario

The flow of this action is represented in figure 4.8

Below is presented the steps necessary, without interruptions, for querying from console a selected diagram.

Happy flow:

1. User opens or creates a diagram
2. Select translating the diagram in abox
3. System returns the racer file containing the abox
4. User loads the diagram
5. User selects the command console

6. User enters query in console

7. System displays the results in the console

In the extended flow are presented the interruptions that might occur when trying to complete this action.

Extended flow:

4.1 Racer engine is not found

4.1.1 System notifies the user that RacerPro was not found
4.1.2 The user starts manually the engine

6.1 Console is not visible

6.1.1 User open window->show perspectives->console
6.1.2 System displays the console view
6.1.3 User selects Command Console from the console view
6.1.4 System displays the command console

6.1 Racer engine stopped working

6.1. System notifies the user that RacerPro was not found
6.2 The user starts manually the engine

6.1 Query is incorrect!

6.1. System notifies in the console the user to re-enter the query
6.2. User re-enters query

4.5.2 Validating the Safety Case

When analysing the diagram by querying the RacerPro engine the safety engineer can simply identify the goals from the diagram that are still undeveloped or not supported by evidence, goal descriptions or retrieve explanation why a goal belongs to a specific concept, check the consistency of the Abox. In this way, the safety engineer can repair the problems and validate.

The following formal verifications are provided by the SafeEd system:

1. Every node can be traced back to the top-level claim. That is, there are no "dangling" nodes or sets of nodes.
Table 4.2: Retrieving information about the safety case.

<table>
<thead>
<tr>
<th>Query</th>
<th>RacerPro query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top level goal</td>
<td>(concept − instances TopLevelGoal)</td>
</tr>
<tr>
<td>Support goals</td>
<td>(concept − instances SupportGoal))</td>
</tr>
<tr>
<td>Evidence supporting goal $g_2$</td>
<td>(retrieve − individual − fillers $g_2$ hasEvidence)</td>
</tr>
<tr>
<td>Undeveloped Goals</td>
<td>(concept − instances UndevelopedGoals)</td>
</tr>
<tr>
<td>Generate OWL</td>
<td>(save − kb”PATH/kb.owl” : syntax : owl)</td>
</tr>
<tr>
<td>Check if Abox is consistent</td>
<td>(abox − consistent?)</td>
</tr>
<tr>
<td>Get all contexts of a specific goal</td>
<td>(individual − fillers $g_1$ inContextOf)</td>
</tr>
</tbody>
</table>

2. Each "leaf" node should either evidence or a reference to some previously reviewed assurance case

3. Circular reasoning: identified by the RacerPro engine in the form of cycle concepts

Validating the Diagram Scenario

The flow of this action is represented in figure 4.9

Below is presented the steps necessary, without interruptions, for validating using Racer the selected diagram.

Happy flow:

1. User opens or creates a diagram
2. User select from ProjectExplorer tab the selected diagram
3. Selects validation of the wanted diagram
4. Systems returns the results in validation.log file

In the extended flow are presented the interruptions that might occur when trying to complete this action.

Extended flow:

3.1 File containing diagram translation is not found
   3.1.1. System notifies the user that racer file was not found
   3.1.2. The user selects translating the diagram
   3.1.3. User re-selects validation

3.2 Racer engine is not started
   3.2.1. System notifies the user that RacerPro was not found
3.2.2. The user starts manually the engine

3.2 Diagram is not loaded

3.2.1. System notifies the user that diagram is not loaded
3.2.2. The user loads the diagram in racer engine

4.5.3 Generation of Safety Case Metrics

Complementarily to supporting semantic reasoning, our system provides also quantitative assessment of a safety case through several metrics developed.
The metrics are developed with the LISP API of RacerPro system. For instance, the number of not verified goals for safety case given as the ABox \( sc_1 \) is computed with:

\[
\text{(length (concept - instances NotVerifiedGoal))}
\]

or the number of undeveloped goals:

\[
\text{(length (concept - instances UndevelopedGoal))}
\]

The main use case of metrics is to assess the progress during different stages of validating the safety case. Given large safety cases, one can monitor the rate to which the number individuals of type \( \text{NotVerifiedGoal} \) decreases.

4.5.4 Generating Natural Language Reports on the Safety Case

Our tool supports the generation of documentation and reports for the safety case. There will be generated three types: validation report, to do or assessment report and diagram documentation.
The to-do reports contains things that still needs to be done, assessment and validation of the safety case. The validation report includes:

- nodes that do not have a description;
- elements that are not linked directly or indirectly through other elements of the diagram to the top level goal;
- goals that do not have evidence or solution;
- incomplete goals that have undeveloped sub-goals.

With this report the safety engineer knows at any moment what still needs to be added to the safety case to have a complete and well-build safety case. The assessment report provides also quantitative information of the diagram, in terms of number of nodes and their types. The documentation file will include the assessment report and description of every goal node of the diagram, including attributes and relations. Having the diagram and diagram documentation facilitate the work of the safety engineer or certification auditors.
Figure 4.8: Flow Chart Querying the diagram
Figure 4.9: Flow Chart for validating the diagram
Chapter 5

Detailed Design and Implementation

Our tool is an improvement of Acedit [1] tool. New features like the option of querying the diagram, generation of reports and documentation, usage of safety metrics for the assessment of the safety cases have been added and others tool features have been improved, for example the validation of a diagram.

5.1 System Architecture

The tool consist of a set of Eclipse plug-ins.

![System Architecture Diagram]

Figure 5.1: System Architecture

As presented in figure 5.1, the tool is structured on layers as follows:

- **first layer**: is the one at the bottom and consists of the core framework of the tool;
CHAPTER 5. DETAILED DESIGN AND IMPLEMENTATION

- **second layer**: consists of several eclipse plug-ins used to implement the tool (EMF, GMF, GEF) plus the libraries used for establishing connections with RacerPro engine and NaturalOwl engine;

- **third layer**: contains the GSN and ARM metamodels, plus tool plug-ins through which all tool functionality is provided;

- **last layer**: represents the user interface layer and consists of the GSN editor and the model management tools (GSN editing wizards);

From figure 5.2 can be seen that we kept the main component of the ACedit, consisting of the GSN plug-ins regarding the graphical editor, eliminated the validation plug-in and we added a new set of plug-ins which provides the semantic reasoning facility, validation and the others features.

![Figure 5.2: Component Diagram](image)

### 5.1.1 GSN editor

The GSN editor has two views:

- the diagram view: deals with creation of GSN diagram accordingly to the mapping between metamodels and their graphical notation

- the model view: deals with creation of GSN metamodels
For communication between these two views, it has been used the Model View Controller pattern as follows: the model part of the MVC is represented by the model view and is the GSN model instance created by the user, the view part is represented by the diagram view, i.e., the editor's GUI, and the controller represents the logic implemented in the other plug-ins. This communication is also represented in Figure 5.3.

Figure 5.3: Component Diagram

5.2 Implementation details

5.2.1 GSN plug-ins

In this set of plug-ins, GSN editor functionality and interface are implemented using specific Eclipse frameworks designed for this:

**Eclipse Modeling Framework (EMF)** is a Java framework and code generation facility provided by Eclipse for building tools based on domain models. The framework provides the EMFatic language with which the domain models can be turned into efficient, correct, and easily customizable code. EMFatic language is similar to Java syntax and is designed to support navigation and editing of Ecore models. In our case, the GSN metamodels are defined using the EMFatic language.

**Graphical Editing Framework (GEF)** is a framework provided by Eclipse for creating graphical editors for various diagrams. It is used to represent graphically a safety case and also permits the addition and manipulation of a safety case, providing a visual representation of the relationships between the safety case model elements.

**Graphical Modelling Framework (GEF)** is an Eclipse framework used for developing graphical editors based on EMF and GEF. This has been used to develop the runtime infrastructure of our tool.
5.2.2 GSN ontology plug-ins

The GSN console plug-in contains the implementation of the console used to run queries on the diagram called Command Console. It consist of two packages, gsn1.console is the package implementing the console and gsn1.racercommands is the one which deals with the processing and execution of the commands.

We create a new type console called "Commands Console" implemented in class CommandConsole. This class extends the IOConsole class provided in org.eclipse.ui.console package. The console will display text from the I/O streams. It can have multiple output streams connected to it and only one input stream connected to the keyboard.

In the plugin.xml file we declare a new extension point that contributes to the org.eclipse.ui.console.consoleFactories factory, extending that plug-in with the console factory class called Factory. The console factory extension is responsible for opening a console in the console view. Extensions appear on a menu in the console view and are opened when their openConsole method is called. The gsn.console package that contains the console was developed so that it can be re-used by developers for building I/O consoles having other purposes.

For processing the commands and queries from the console we applied the Factory pattern. There is the factory class called CommandFactory and the interface ICommand implemented by each command. The StartCommand class deals with starting a new RacerPro process if at that moment there’s no process of the engine. The QueryCommand class is used to send a query to the racer engine and process the results.

The class diagram of this plug-in can be seen in figure 5.4.

The GSN actions plug-in adds in the diagram's file menu all the actions that can be done with the file: transforming into A-box, validate it, load it in the racer engine, generate to-do report and documentation. It is composed of two packages, one is containing the popup actions and the second one contains their core functionality of the popup actions from the menu. The actions have a corresponding class from the gsn.actions package and action in the extension point extending the org.eclipse.ui.popupMenus class from the plugin.xml file in order to be visible in the view. Their core functionality has been separated from the interface, all classes being implemented in the package gsn.actions.core. All the 3 types of actions for creating reports use the same class DocumentationActionCore which has a separated method for each one of them because they don’t have the same template. All the others actions are linked to a class implementing the action functionality.

In figure 5.6 are represented all the classes and the relations between them from this plug-in.

The GSN parser plug-in is the eclipse plug-in used to translate the builded safety case diagram into the A-box part. We have a class called XMLparser that has methods for parsing each node from the diagram-model file( which is a xml file) transforms it into a concept instance or relation between two instances or adds an attribute and at the end saves all data into a file representing the an A-box. Because each node of the diagram is
5.2. IMPLEMENTATION DETAILS

Figure 5.4: Class diagram for console plug-in.

represented through generated ids at not by the node identifier from diagram, when we parse the diagram model file create Shape objects and save only the nodes that are shape or shape attribute, if we find connectors that depending on the ids representing the targets we search for those shapes and add the connection at the parent shape. This class uses org.w3c.dom and javax.xml.parsers libraries for parsing the XML file.

The Ontology class is contains methods that build the syntax to be added in the Abox file, depending on the node type.

In figure 5.6 are represented all the classes and the relations between them from this plug-in.

The GSN ontology plug-in is implementing the communication of our tool with the RacerPro engine.

The class called OntologyConnection is build following Singleton pattern, and is used to create a racer client and establish a connection to the racerPro reasoner using the JRacer java library provided by Racer.

For creating the connection we need to check if the engine is started, for this we have the class RacerProcess which checks if there exist any process of the engine, if not then locates the execution file of the racer and starts a new process. If no file has been found then the user will be shown a message with the location from where he can download the engine.

This OntologyConnection class makes it possible interrogating the build A-box file, mainly by receiving the text command from the console and calling answerQuery(String
command) method which will return the result of the sent query. Also this class has the
method getTboxFile() which instantiates an GSNTbox object and asks for creating the
t-box file modeled on GSN standard.
5.2. IMPLEMENTATION DETAILS

The GSNTbox class is a class that builds the T-box part of the ontology. It has static methods where all the concept, roles and attributes are defined, returning list of strings that are merge in the getContent() method of the class. A new file is generated with all this info and it will be saved at running in the eclipse main folder.

In figure 5.7 are represented all the classes and the relations between them from this plug-in. This plug-in is used in all the other plug-ins excepting the parser plug-in.

The GSN reporting plug-in contains three types of reports: validation, ToDo and documentation reports generated in natural language. All three reports are pdf type and are generated using iText library. This library is used to create, adapt, inspect and maintain pdf files.

The data from the reports is taken using Lisp Racer Cilent library provided by Racer is used for generating metrics in the reports and by directly interrogating the A-box with fixed queries depending on each report type. Each report has a different template.

The validation report contains the same output of the validation action, the difference is that it is saved in a new file and not in a log file.

The To do report contains what has been done, what still needs to be done and what is missing from the safety case diagram.

Figure 5.6: Class diagram for parser plug-in.

The GSNTbox class is a class that builds the T-box part of the ontology. It has static methods where all the concept, roles and attributes are defined, returning list of strings that are merge in the getContent() method of the class. A new file is generated with all this info and it will be saved at running in the eclipse main folder.

In figure 5.7 are represented all the classes and the relations between them from this plug-in. This plug-in is used in all the other plug-ins excepting the parser plug-in.

The GSN reporting plug-in contains three types of reports: validation, ToDo and documentation reports generated in natural language. All three reports are pdf type and are generated using iText library. This library is used to create, adapt, inspect and maintain pdf files.

The data from the reports is taken using Lisp Racer Cilent library provided by Racer is used for generating metrics in the reports and by directly interrogating the A-box with fixed queries depending on each report type. Each report has a different template.

The validation report contains the same output of the validation action, the difference is that it is saved in a new file and not in a log file.

The To do report contains what has been done, what still needs to be done and what is missing from the safety case diagram.
5.3 User Interface

The workspace of the system is presented in Fig. 5.8. A safety project (top-left) consists of several assurance cases, developed either as a graphical diagram (files with gsn extension) or as an abox in description logic (files with racer extension). In case of need the system automatically translated between these two input formats. For a selected diagram file the user can transform into abox, validate the diagram and generate reports.

The main window (top-center) depicts the active gsn diagram. The elements of the GSN standard are represented as follows: goals with rectangular, strategies with parallelograms, evidence and solutions are represented by circle, assumptions and justifications with ellipse, context by a rectangular with rounded corners, the supportedBy relation is an arrow with the head filled, while the inContextOf is represented by an arrow with empty head.

The title and description of a node can be entered by clicking on the node in the head part for the title, and in the field with the placeholder ‘description’. The diagram is constructed by using a drag-and-drop pallet (top-right).

The built-in actions performed on the diagram are visible when clicking right on the diagram or model file.

The command console (bottom-center) shows the reasoning performed on the active
diagram above. In the command line, specific queries for interrogating ontologies can be added and the reasoning engine will return the results for each query. The syntax of the queries corresponds to the RacerPro tool. In Fig. 5.8, the four queries exemplified are:

- retrieving all the goals in the diagram,
- identifying the top level goal,
- listing all pieces of evidence supporting the goal $g_2$, and
- checking the consistency of a diagram with respect to the GSN standard encoded as axioms in description logic.

In the bottom-left corner the red rectangle represent the view part of the diagram visible in the main window.
Figure 5.8: Application Interface
Chapter 6
Testing and Validation

6.1 Testing the tool

To test if the application has the desired behaviour and it is well build, we have taken
well structured or incomplete safety cases from different domains, built and validated
within our tool. More details about each safety case can be found in the Bibliographic
research chapter.

The first safety scenario is taken from the autonomous driving domain and
represents in the case is claimed that any autonomous vehicle should ensure safety when
operating in the environment. A GSN diagram built in our SafeEd tool is represented in
Fig. 6.1.

The top level goal $g_1$ states that any autonomous vehicle should ensure safety when
operating in the environment. The goal holds in two contexts: the existence of an environ-
ment formalisation (context $c_1$), respectively the existence of a mechanism providing situ-
atuation awareness. One solution for ensuring safety is dynamic risk assessment approach [9].
The corresponding strategy $s_1$ used to support the goal $g_1$ is to dynamically assess the
risk. The sub-goals $g_2$, $g_3$, $g_4$, and $g_5$ are used to fulfill the strategy $s_1$. For instance, the
sub-goal $g_2$ claims the correctness of the model, statement that is supported by various
pieces of evidence, including formal verification $e_2$

The diagram from Fig. 6.1 is translated into the description language in Figure 6.2
and the result of translation into Abox using Racer syntax by our tool is shown in figure
Figure 6.3. Here, the facts $f_{61}$ to $f_{64}$ assert the individuals to their corresponding GSN core
elements. The structure of the GSN diagram based on the two relationships supportedBy
and inContextOf is formalised by the facts $f_{65}$ to $f_{72}$. The natural language text describing
claims, solutions, contexts or evidences are encapsulated as concrete attributes [5] in Racer
syntax (assertions $f_{73}$ to $f_{80}$).

The table 6.1 contains the results of some basic queries run on the abox from 6.2. It
can be seen that the obtained results are very helpful because it spares the safety engineer
time which has to look for manually for them if such an assessment feature, like querying
the diagram, is not available.
Table 6.1: Retrieving some information about the safety case represented in figure 6.1.

<table>
<thead>
<tr>
<th>Query</th>
<th>RacerPro query</th>
<th>RacerPro answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top level goal</td>
<td>(concept – instances TopLevelGoal)</td>
<td>g_1</td>
</tr>
<tr>
<td>Support goals</td>
<td>(concept – instances SupportGoal))</td>
<td>g_2, g_3, g_4, g_5</td>
</tr>
<tr>
<td>Evidence supporting goal g_2</td>
<td>(retrieve – individual – fillers g_2 has – evidence)</td>
<td>e_1, e_2, e_3, e_4</td>
</tr>
<tr>
<td>Undeveloped Goals</td>
<td>(concept – instances UndevelopedGoals)</td>
<td>g_3, g_4, g_5</td>
</tr>
<tr>
<td>Check if Abox is consistent</td>
<td>(abox – consistent?)</td>
<td></td>
</tr>
<tr>
<td>Get all contexts of a specific goal</td>
<td>(individual – fillers g_1 inContextOf)</td>
<td>c_1, c_2</td>
</tr>
</tbody>
</table>

The second safety case is taken from medical industry and refers to the Therac 25, a case study in safety failure. In figure 6.4 is a diagram build wit our tool representing the case of Therac 25 presented in [12].
6.1. TESTING THE TOOL

Figure 6.2: Translating the GSN diagram in a description logic Abox.
Figure 6.3: Translating the GSN diagram in Abox using our tool.

The main statement is represented by top level goal \( \text{Goal}_1 \) and states that the machine is fault free and can be used. Two strategies have been adopted for proving this goal: the first strategy represented by \( \text{Strategy}_1 \) proposes argumentation by satisfaction of system’s safety, while the second strategy represented by \( \text{Strategy}_2 \) proposes argumentation by omission of all identified software hazards supposing we have identified the software hazards, context represented by node \( \text{Context}_1 \). The first strategy is divided into two sub-claims: the system will halt in case of errors, statement represented by \( \text{Goal}_2 \), and that exceeding radions limit will abort the operation, statement represented by \( \text{Goal}_3 \). \( \text{Goal}_2 \) is supported by \( \text{Goal}_4 \) which claims that when a fault occurs the system cannot be resumed, a solution to this would be Black Box testing(\( \text{Solution}_1 \)) Both \( \text{Goal}_2 \) and \( \text{Goal}_3 \) are supported by the \( \text{Goal}_5 \) which claims that any failure operation will include explanatory error message, a solution for this is declaring and using safe states(\( \text{Solution}_2 \)). The
6.1. **TESTING THE TOOL**

The second adopted strategy is solved by two statements: computer giving a wrong amount of energy occurs as a result of component failure, \(\text{Goal}_6\), and computer selecting wrong mode can only occur as a result of component failure, \(\text{Goal}_7\). Both goals are solved by: making fault tree analysis, \(\text{Solution}_3\) and analysing the results of hazard test, \(\text{Solution}_4\).

This case could have been applied when developing the Therac 25 system, leading to discovering of several faults that could have saved lives.

The diagram in Fig. 6.5 is translated into the Abox represented in Fig. 6.2. Here, the facts assert the individuals to their corresponding GSN core elements. The structure of the GSN diagram based on the relationships has – evidence, supportedBy and inContextOf. The natural language text describing claims, solutions, contexts or evidences are encapsulated as concrete attributes [5] in Racer syntax.

The table 6.2 contains the results of some basic queries run on the abox from 6.5. It can be seen that the obtained results are very helpful because it spares the safety engineer time which has to look for manually for them if such an assessment feature, like querying the diagram, is not available.

<table>
<thead>
<tr>
<th>Query</th>
<th>RacerPro query</th>
<th>RacerPro answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top level goal</td>
<td>(\text{concept} - \text{instances TopLevelGoal})</td>
<td>\text{goal}_1</td>
</tr>
<tr>
<td>Support goals</td>
<td>(\text{concept} - \text{instances SupportGoal}))</td>
<td>\text{goal}_2, \text{goal}_3, \text{goal}_4, \text{goal}_5, \text{goal}_6, \text{goal}_7</td>
</tr>
<tr>
<td>Evidence supporting goal \text{goal}_7</td>
<td>(\text{retrieve} - \text{individual} - \text{fillers strategy}_2 \text{inContextOf})</td>
<td>\text{solution}_3, \text{solution}_4</td>
</tr>
<tr>
<td>Undeveloped Goals</td>
<td>(\text{concept} - \text{instances UndevelopedGoals})</td>
<td>\text{nil}</td>
</tr>
<tr>
<td>Check if Abox is consistent</td>
<td>(\text{abox} - \text{consistent}?)</td>
<td>\text{t}</td>
</tr>
<tr>
<td>Get all contexts of a specific goal</td>
<td>(\text{individual} - \text{fillers strategy}_2 \text{inContextOf})</td>
<td>\text{context}_1</td>
</tr>
</tbody>
</table>
Figure 6.4: Therac 25 safety case.
6.1. TESTING THE TOOL

The third safety case is for a whole-airspace system divided into several geographical regions plus a region of en-route airspace for which ATM rules are applied and implemented on an area-by-area basis. Geographical regions that interact with each other and also with the airspace-wide system. The safety case structure for the system is represented in figure 6.6. The main goal represented as Goal1 states that the airspace is safe for using, in the context of defining the safe term in case of the system. The strategy adopted by proving the goal is to prove that ATM rules are safe, respectively that the rules have been implemented safely. The second strategy is also divided in two: it is state that the rules are implemented safely in every area and each area assumption can’t be violated. Evidence for each sub-goal is provided, as is shown in the diagram.

Figure 6.6 represents the translated Abox into Racer syntax, it can be seen that the goal structure is keep in the A-box.

<table>
<thead>
<tr>
<th>Query</th>
<th>RacerPro query</th>
<th>RacerPro answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top level goal</td>
<td>(concept – instances TopLevelGoal)</td>
<td>g1</td>
</tr>
<tr>
<td>Support goals</td>
<td>(concept – instances SupportGoal))</td>
<td>g2, g3, g4, g5, g6, g7</td>
</tr>
<tr>
<td>Evidence supporting</td>
<td>(retrieve – individual – fillers</td>
<td>e3</td>
</tr>
<tr>
<td>goal</td>
<td>goal7 has – evidence</td>
<td></td>
</tr>
<tr>
<td>Undeveloped Goals</td>
<td>(concept – instances UndevelopedGoals)</td>
<td>nil</td>
</tr>
<tr>
<td>Check if Abox is consistent</td>
<td>(abox – consistent?)</td>
<td>t</td>
</tr>
<tr>
<td>Get all contexts of a specific goal</td>
<td>(individual – fillers strategy2 inContextOf)</td>
<td>nil</td>
</tr>
</tbody>
</table>
In table 6.4 we presents the conclusion after building the safety cases presented above in our tool and two other tools. More detailed comparison between our tool and other tool can be found in Bibilographic Research Chapter. From the table we can resume that our tool is better at the management of a safety case, having a plus at validation,
6.1. TESTING THE TOOL

Figure 6.7: Airspace-system safety case translated in Abox using our tool

assessent, evaluation and reports generation. A minus would be the weak graphically representation of elements and importing/exporting diagrams from our tool.

The difference between our tool and this one is the fact that using ours the user will be able to build simple or complex queries for interrogating the diagram at any time during the development and not being limited only to metrics, this is a plus at the assessment of the diagram.
Table 6.4: Tools Comparison Table

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Acedit</th>
<th>AdvoCATE</th>
<th>Our Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating diagram</td>
<td>GSN models must be differentiated better in the interface</td>
<td>GSN elements are better represented (color is a plus)</td>
<td>GSN models must be differentiated better in the interface</td>
</tr>
<tr>
<td>Edit diagram</td>
<td>can edit only diagrams build with the tool</td>
<td>can edit other diagrams type</td>
<td>can edit only diagrams build with the tool</td>
</tr>
<tr>
<td>Import/Export other diagram types</td>
<td>can’t import other diagrams types</td>
<td>can import other diagrams types from: D-case, Asce</td>
<td>can’t import other diagrams types</td>
</tr>
<tr>
<td>Metrics</td>
<td>not available</td>
<td>Limited computation of metric</td>
<td>available using Racer lisp library</td>
</tr>
<tr>
<td>Diagram Assessment and evaluation</td>
<td>not available</td>
<td>all diagram nodes are saved on columns in CSV based on their type than limitation of displaying on columns</td>
<td>Reasoning services used. Better, dynamic and more flexible solution</td>
</tr>
<tr>
<td>Diagram Validation</td>
<td>based only on constraints</td>
<td>based only on constraints</td>
<td>based only on constraints plus diagram consistency and missing elements</td>
</tr>
<tr>
<td>Reporting</td>
<td>not available</td>
<td>CSV with nodes structured on columns</td>
<td>Three pdf reports: validation, to-do and documentation</td>
</tr>
<tr>
<td></td>
<td>available</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7

User’s manual

7.1 System Installation

For using the tool, the user needs to have installed on the PC the following:

- Eclipse Distribution with Epsilon, this can be downloaded from here:
  http://www.eclipse.org/epsilon/download/
- RacerPro knowledge base reasoner on 32 bits
- NaturalOwl engine

To install the tool through eclipse updates:

1. Open the wizard and select from Eclipse top menu Install/Update->Available Software Sites
2. Click the Add.. button
3. If the software site is in your local file system click Local.. to specify the directory location of the file.
4. If the software site is in your local file but packaged as a jar or zip, click Archive... to specify the name of the file.
5. Select Add Site...
6. Select all checkboxes
7. Click next
8. Accept Software license
9. Click finish
7.2 User’s Manual

Creating a project and add diagram to it - steps:

1. select File->New->Project
2. select General-> Project
3. select next and give a name to your project
4. press right click in the created project
5. select New->Other
6. in the pop up wizard type GSN
7. select GSN1 diagram
8. select Next-> Finish

This action will add a new project to the workspace and the diagram in it.

Transform diagram in A-box( after diagram file is created):

1. select diagram file
2. press right click on files
3. select Ontology
4. next select Transform A-box

The output of this action will be a new file having the same name as the diagram and the extension .racer located in the same project folder as the diagram. The file is not loaded in the Racer engine!

Load diagram ( only if the racer file has been generated):

1. select diagram file
2. press right click on files
3. select Ontology
4. next select Load A-box

This action adds the A-box to the T-box in the Racer and will set it as main A-box.

Validate diagram:

1. select diagram file
2. press right click on files
3. select Ontology
4. next select Validate A-box

The action will write the results of the validation in validate.log file, this file will also contain the results of the previous validations of this or other files. An example of this report is shown in figure 7.1.

```
All the nodes have claims!
Evidence or solution must be provided for the following goals:
G3, G4, G5
G3 goal parents: G1
G4 goal parents: G1
G5 goal parents: G1

G1 goal has undeveloped childs
All goals are linked to the top-level goal!
```

Figure 7.1: Autonomous vehicle scenario.

Generate Validation report:
1. select diagram file
2. press right click on files
3. select Reports
4. next select Validation report

The action is similar to validation action, the difference is that the report contains info only about this file.

The generation of documentation and todo reports is the same as this. The generated file are of pdf type.

Generate ToDo report:
1. select diagram file
2. press right click on files
3. select Reports
Generate Documentation:

1. select diagram file
2. press right click on files
3. select Reports
4. next select Documentation

The user can query a diagram by following the next steps:

1. if the command-console is not visible then add it in the view
   - select window->perspective->other
   - select console from the view

2. Write `start` and press enter to activate the console

3. select Ontology->Load Abox (to be sure that the diagram si loaded)
   or write `(set – current – abox diagramName)`

4. write the query and press enter

The console will look like this:

![Command Console](image)

Figure 7.2: Command Console.
Chapter 8

Conclusions

8.1 Contributions and Achievements

From this paper we can deduce that it is very necessary to be sure that a system has a correct behavior and can be operately successfully so it is very important to build a safety case that describes the real level of safety that the system assures.

Assurance cases have evolved from the concept of safety cases being a requirement in many international standards. It is very important to build well-structured and coherent safety cases because wrong construction and reasoning in safety arguments can undermine a systems safety claims and lead to failures of the system. Like in case of Therac25, not building a simple assurance case (that would have lead to discover several faults) resulted in death of people.

All the elements of a safety case are part of the causal chain of hazard. The safety case is based on claims, any claim is broken into sub-claims until is reached a point when the claims can be proved by the development or assessment of an artefact as evidence. Every claim and strategy adopted to support the claim should be very clearly formulated and state the context in which the argument is made so that the graphical representation of the safety case will be well structured.

It’s a good practice to develop the safety cases prior to development because in this way you find the risks and faults before starting the development so you’ll have the solution to mitigate them.

This paper described our tool that can not only be used to build safety arguments according to the Goal Structuring Notation standard but it also provides a better management and assessment of the safety case. The novelty of our approach is that the assurance case is automatically translated in description logic. The advantage is that the specific reasoning services of description logic are enacted to verify the compliance of the case with the GSN standard and also to signal possible argumentation flaws. The tool was demonstrated when developing safety all the safety cases presented in Chapter Testing and Validation. The main advantage of our the tool is that it can reduce the risks of not building well-structured safety cases or not provide evidence for all the statements from
the safety case.

Another big advantage is that our tool is extensible and can integrate with other corporate applications developed based on the Eclipse platform. In this line, ongoing work regards enhancing the tool with other.

8.2 Further Work

There are some improvements that can be done in the future to improve the tool and add new functionality. Being build as a set of eclipse plug-ins that need the eclipse platform to run on, the biggest improvement would be to get rid of this and run the tool as a standalone application by creating a Rich Client Platform (RCP) application for the tool. We choose this because it will improve the portability of the tool. RCP can run as standalone applications by including a minimal set of eclipse plugin Other future improvements:

- reverse the process of translation, from an A-box build accordingly to the T-box the user should be able to transform it into diagram. This will ease a lot more the work of the safety engineer.
- import and export diagrams from other editors
- improve the user interface: the metamodels should be colored distinctly from the start
- the generated reports should provide more information
- add the possibility to extend the metamodels
Bibliography


69
Appendix A

Published papers
