An Approach to Automatic Generation of Evidence for Safety Cases

Master of Science Project at DaimlerChrysler

Willem C. Ridderhof
An Approach to Automatic Generation of Evidence for Safety Cases

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Willem C. Ridderhof
born in Tholen, The Netherlands

Delft University of Technology
Software Engineering Research Group
 Faculty EEMCS
 Mekelweg 4
 2628 CD Delft, The Netherlands
 www.ewi.tudelft.nl

DaimlerChrysler
DaimlerChrysler AG
Research Berlin
GR/ESM
Alt-Moabit 94a
10553 Berlin, Germany
www.daimlerchrysler.com
© 2006 Willem C. Ridderhof.

Document generated with \LaTeX

Cover picture: Mercedes Benz Actros Safety Truck (DaimlerChrysler TIMES, May 2006)
An Approach to Automatic Generation of Evidence for Safety Cases

Author: Willem C. Ridderhof
Email: willem.ridderhof@daimlerchrysler.com

Abstract

Safety critical systems are systems that can kill or hurt people or can cause serious environmental damage. To protect people for losses caused by such systems, regulation authorities have created safety standards. Contemporary safety standards describe a set of best practices on how to develop a safe system.

Currently, a new safety standard for the automotive industry is being constructed. This standard has been titled "ISO/WD 26262 Road vehicles: Functional safety". The ISO/WD 26262 is still a working draft which implies that present-day systems do not have to comply to the standard, yet. To be ready for the introduction of the final version of the standard, the newly introduced concept of the Safety Case has been studied. The results of this study will be presented in this thesis.

A Safety Case provides an argument why a particular system is acceptably safe. A Safety Argument consists of certain claims and evidence for those claims. This thesis shows which part of the evidence can be generated automatically. It turned out to be possible to automatically generate evidence for a traceability part of the Safety Argument. To do so, a traceability part of the Safety Argument has been constructed. Based on this argument, a method has been specified that defines how to generate traceability evidence. Finally, a tool has been created to automatically generate the evidence.

Keywords: Emergency Brake System, Goal Structuring Notation, ISO/WD 26262, Object Constraint Language, Safety Argument, ToolNet

Thesis Committee:

Chair: Prof. Dr. Ir. Arjan J.C. van Gemund, TU Delft
University supervisor: Dr. Phil. Hans-Gerhard Gross, TU Delft
Company supervisor: Dr. Heiko Dörr, DaimlerChrysler AG
Committee Member: Ir. Bernard R. Sodoyer, TU Delft
preface

Without the support of enthusiastic colleagues, I would not be able to write this thesis. I like to thank the ToolNet team, consisting of Daniel Dienhardt, Jürgen Grossmann René Freude, Andreas Pasch, and Anna Trögel. I especially have to thank Andreas and Anna for their support by the creation of the Solution Generator and Daniel, Jürgen, and René because they were always willing to discuss about the concepts I needed and invented and they all helped me to get familiar with ToolNet.

Furthermore, I thank Jürgen Bohne and Reny Grönberg for their advise about safety topics. Besides that, Jürgen was willing to review parts of my thesis. Kerstin Buhr and Hartmut Pohlheim for introducing me into the area of development methods. They showed me how contemporary systems at DaimlerChrysler are being developed. I thank Matthias Hoffmann and Matthias Weber for their information about requirements engineering and Christian Scheidler who introduced me to the EASIS project, which gave me contacts to numerous documents about safety. I also thank Michael Amann from ZF Friedrichshafen AG, which reviewed key elements of my thesis.

Last but certainly not least, I really like to thank Heik Dörr, Gerd Gross, and Arjan van Gemund. Without them, my research at DaimlerChrysler would be impossible.

Besides that, I have to thank the people that did not contribute to this thesis directly, but made my stay in Berlin a pleasure. Personally, I am happy that I got the opportunity to stay in Germany for the course of my thesis project. The cultural interchange was exciting. Besides that, the topic of this research was interesting. I hope that my project may contribute to the creation of safer systems!

Willem C. Ridderhof
Berlin, Germany
November 28, 2006
Contents

Preface iii
Contents v
List of Figures ix
List of Tables xi
List of Algorithms xiii

1 Introduction 1
  1.1 When Safety Cases May be Useful ......................... 2
  1.2 The Safety Case in its Context .......................... 3
  1.3 Problem Statement ....................................... 5
  1.4 Summary and Outline of This Thesis ....................... 6

2 Technological Context 9
  2.1 The Safety Case ........................................... 9
      2.1.1 Safety Argument ..................................... 11
  2.2 Traceability .............................................. 14
      2.2.1 Realization of Requirements .......................... 14
      2.2.2 Development Objects .................................. 19
      2.2.3 Creating Associations Between Development Objects ... 21
  2.3 Summary .................................................. 23

3 A Method to Evaluate Safety Cases 25
  3.1 Evaluating a Safety Case ................................. 25
  3.2 Product Model ............................................. 27
      3.2.1 Development Data ..................................... 27
      3.2.2 The Safety Argument .................................. 27
      3.2.3 Solutions ............................................... 34
CONTENTS

3.3 Process Model ........................................... 36
  3.3.1 Creating ToolNet Models ......................... 37
  3.3.2 Creating Solutions ............................... 39
3.4 Summary ................................................. 39

4 A Tool to Generate Solutions 41
  4.1 Requirements ........................................ 41
  4.2 Design ............................................... 43
    4.2.1 A Technique to Create Solutions ............... 43
    4.2.2 The Design Made With OCL ...................... 48
  4.3 Integration and Implementation ...................... 49
  4.4 Summary ............................................. 51

5 Evaluation 53
  5.1 EBS Case Study ................................... 53
  5.2 Evaluation of the Method .......................... 55
  5.3 Evaluation of the Solution Generator ............. 59
    5.3.1 Creating OCL Expressions .................... 60
    5.3.2 Analysis’s Results ............................. 61
    5.3.3 Consistency ..................................... 62
    5.3.4 Scaleability ................................... 63
    5.3.5 Modifiability .................................. 65
    5.3.6 Correctness and Completeness .................. 65
  5.4 Summary ............................................. 66

6 Conclusion 67
  6.1 Summary ............................................ 67
  6.2 Conclusions ....................................... 68
  6.3 Future Work ....................................... 69

Bibliography 73

Glossary 77

Index 81

A The Goal Structuring Notation 83

B Extended Evaluation Method 85
  B.1 Extended Safety Argument ......................... 85
    B.1.1 The Evaluation Method ........................ 87
  B.2 Traceability Report ................................ 88
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Possible decomposition of a safety claim</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Overview of the relation between the main elements of Toulmin’s argumentation</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>An example definition of a Safety Argument specified in the GSN</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Basic decomposition of a Safety Argument</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Process development model of an embedded safety related system at DaimlerChrysler</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>An example development object model (termed IM) for the requirements engineering phase</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Relations between tools, models, and data in ToolNet</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>An example Tools For Projects Model</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td>Traceability overview</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>High level method overview</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>An initial decomposition of a Safety Argument</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustration of traceability directions</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>A decomposition of the origin goal</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>A part of the decomposition of the realization goal</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Process steps needed to evaluate a Safety Argument automatically</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Create a PIM</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Environment of the Solution Generator</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Architecture of ToolNet</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Component decomposition of the Solution Generator</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>The Solution Generator in the ToolNet environment</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>A screenshot of a part of the Navigation Client</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>The EBS in its environment</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Illustration of traceability associations for safety requirements and safety concepts</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Figures

5.3 Illustration of data from the requirement specification phase .......................... 56
5.4 A Safety Argument for EBS ............................................................................. 57
5.5 The IM for EBS ............................................................................................... 58
5.6 The Tools For Project Model for a Safety Requirement ................................. 59
5.7 A trace table shown in ToolNet’s Navigation Client ............................................. 62
5.8 Comparison of the execution time of the Solution Generator’s components 63
5.9 Time it takes to get the exports from the data sources and instantiate them in the IM MDR ................................................................. 64

A.1 Overview of symbols used in the GSN ............................................................... 83
B.1 Relation between development objects and reviews ............................................. 86
C.1 Levels in the MOF ............................................................................................ 98
C.2 Distribution of IM, Tool Model, development data, and real subsystems over MOF levels ......................................................................................... 98
C.3 A simplified Safety Argument ......................................................................... 99
C.4 A MOF metamodel for the Safety Argument .................................................... 101

D.1 The Safety Analyzer as integrated ToolNet component .................................. 103
D.2 The OCL menu ................................................................................................ 104
D.3 Configuration of the Safety Analyzer ............................................................... 105

E.1 An example evaluation result generated by a Project Adviser .......................... 108
E.2 Colored Safety Argument .................................................................................. 109
<table>
<thead>
<tr>
<th>Table No.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Safety Integrity Levels</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>The steps the GSN construction method</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Tools used during the development process at DaimlerChrysler</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Generic trace table for $\forall a \rightarrow \exists b$</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Extended trace table for $\forall a \rightarrow \exists b$</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Example trace table for $\forall h \rightarrow \exists sg$</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Requirements for the Solution Generator</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Trace table for $\forall h \rightarrow \exists sg$</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Lookup table from patterns specified in the Safety Argument to OCL expressions</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Decomposition of EBS’s requirements</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>The name of associations between development objects as specified in the EBS’s IM</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Lookup table for patterns of logical expression specified in EBS’s Safety Argument</td>
<td>60</td>
</tr>
<tr>
<td>B.1</td>
<td>Example table for the result attribute from Reviews</td>
<td>87</td>
</tr>
<tr>
<td>C.1</td>
<td>Example set of development objects</td>
<td>92</td>
</tr>
<tr>
<td>C.2</td>
<td>Sets of associations between development objects</td>
<td>92</td>
</tr>
<tr>
<td>C.3</td>
<td>Logical equations and corresponding Prolog clauses</td>
<td>94</td>
</tr>
<tr>
<td>C.4</td>
<td>Development data and references represented in the form of tables of a relational database. Primary keys are underlined. When an entity is both a primary and foreign key, that entity has been double underlined</td>
<td>95</td>
</tr>
<tr>
<td>C.5</td>
<td>A trace table generated with OCL</td>
<td>95</td>
</tr>
</tbody>
</table>
List of Algorithms

4.1 Pseudo algorithm to create a IM MDR .......................... 51
Chapter 1

Introduction

Since the beginning of the Industrial Revolution, more and more machines have been used to perform tasks which humans do not want to do, simply cannot do, or which can be done more cost effective by machines. Today, machines—or in general, systems—are getting larger and techniques from various specialist areas are used to build them. An increasing number of systems are being controlled by software, a few out of many examples are mobile telephones or satellite navigation systems. The combination of more specialist areas, larger systems and the introduction of systems controlled by software, causes that systems become more and more complex.

People rely more and more on increasingly complex systems. An example is the Therac-25, a computer controlled radiation therapy machine [31]. The system was used for quite a long time to heal cancer patients. However, the system gave at least six people a massive overdose of radiation which made them suffering more and made that they died more quickly.

Safety critical systems are systems that may cause harm or damage to humans or the environment when malfunctioning [37]. According to this definition, the Therac-25 is a safety critical system. The difficulty is that it is hard for end users to determine whether a system is safe or not. Leveson [30] claims that the Western society of today tends to shift responsibility on safety issues from the end users of a product to the manufacturer. This is, among others, caused by the increasing complexity and scale of todays technological systems and, besides that, the pace of technological change is increasing, too. Most of the end users cannot understand the systems anymore and have to assume that the manufacturer takes care of the safety of the system.

This shift in responsibility means that the user should trust that a manufacturer’s system is safe. However, how does a customer know whether a system is safe? Leveson defines safe as the freedom from accidents or losses, i.e., absolute safe. However, how can it be proved that a system is absolute safe and, furthermore, is absolute safety relevant? The Therac-25 example makes clear that it is difficult, if not impossible, to prove that a system is absolute safe. Although absolute safety has not been proven in advance, physicians and patients were willing to use the
Leveson’s definition of safety seems to be too strict. Users tend to accept systems of which absolute safety has not been proven a priori. This has been clearly defined by the safety standard for Programmable Electronic Safety-Related Systems [10]. It defines safety as the freedom from unacceptable risk.

The term “acceptably safe” has been worked out in safety standards. The IEC 61508 [10] is a well-known safety related standard for programmable electronic devices in general. A new standard, specifically meant for the automotive industry, is currently being developed by the ISO. The working draft, named ISO/WD 26262 [38], will become one of the most important standards for the automotive industry in the future. To verify that manufacturers develop systems in accordance with the relevant safety standards, regulation authorities order specific institutions to carry out inspections. An important inspecting agency for the automotive industry in Germany is the Technischer Überwachungsverein [45]. When an inspecting agency decides that a system fulfills the standards, the manufacturer is allowed to bring the product on the market. See [42] for an example assessment report written by the Technischer Überwachungsverein (TÜV).

In summary, the responsibility for the safety of systems has shifted from the user of a system to the manufacturer. In the automotive industry, manufacturers have to deal with increasing responsibility for safety issues. As a consequence, they need to introduce mechanisms in the development process that show that a system is acceptably safety. Most of these mechanisms are prescribed by the safety standards. The mechanism which will be studied in this thesis is the Safety Case. A Safety Case is a document that provides a clear argument why a system is supposed to be acceptably safe. Such an argument consists of a set of premises and evidence for those premises. This thesis will describe which part of the evidence can be generated automatically and how this can be done.

Section 1.1 will provide an example that illustrates the advantage of using Safety Cases. Then, Section 1.2 shows how Safety Cases are used in their context. In Section 1.3, the contribution of this thesis will be described and finally a summary and an outline of this thesis will be given in Section 1.4.

### 1.1 When Safety Cases May be Useful

At the end of the 1980s, Audi—a car manufacturer—was sued in the United States of America [24]. Some drivers who drove cars produced by Audi, reported that some of these cars suddenly accelerated. When this happened, the drivers reported, the car could not be stopped anymore, even if they rammed on the brake. A six-year-old child had been killed in an accident that looked like just such an accident.

Several people, who all claimed to have similar accelerating problems with their cars, went to court. They claimed that there was an unidentified defect in the design of the Audi 5000. Some courts declared Audi guilty, whereupon it had to pay a lot of reimbursement. However, as had been found out by the National Highway Traffic
Safety Administration, the Audi 5000 was not defective at all. The cause of sudden acceleration was the foot of the driver placed accidentally on the wrong pedal! As may be clear, these litigations were bad for Audi’s reputation.

As stated before, Safety Cases give a clear argument why a system is acceptably safe. When the argument is clearly structured and provides valuable, trustworthy evidence, it is highly likely that most people agree that the system is safe.

Assume Audi had had a Safety Case for its Audi 5000, then it might have been easier to convince judges and juries that the car was safe. The argument of the Safety Case could consist of a list of identified hazards. An example of a hazard referenced in the argument can be “The car accelerates without the gas pedal being pressed”. Based on this hazard, one or more safety requirements can be formulated that prevent this hazard from happening. Finally, evidence should be given by test results which indicate that it is unlikely that the system prevents that the hazard occurs. This in turn shows that the system is acceptably safe with respect to this hazard.

As can be concluded from the example, Safety Cases should not only be generated to satisfy safety standards, but they can also be used in court to convince judges and juries that a particular system is acceptably safe. This, in turn, may save a lot of money and protect the reputation of an enterprise.

1.2 The Safety Case in its Context

The project that provides the basis for this thesis has been carried out at Daimler-Chrysler AG, Berlin, Germany. DaimlerChrysler AG, or simply DaimlerChrysler, is a company that manufactures cars of the brands Mercedes-Benz, Chrysler, etc. Trucks and vans of the brand of Mercedes-Benz are manufactured, too. A lot of research is performed by DaimlerChrysler. Continuously, new systems are developed that make their automobiles the most competitive ones in the market.

Some systems that are used in the automotive industry are safety critical. An example is the cruise control system. This system makes it possible for the driver to maintain a constant speed without the need to press the gas pedal. However, when a cruise control system in a car is malfunctioning a dangerous situation can occur. Assume that the cruise control could not be unlocked anymore and that, by mistake, the brake does not have priority over the cruise control. Then, it would be nearly impossible to stop the car in a safe way because the car is driving at a constant speed and cannot be slowed down anymore.

During this master’s project, a typical driver assistance system has been analyzed which will be referred to as the Emergency Brake System (EBS). The EBS can be compared with the system described in [14]. The EBS is a system that is meant for trucks and its objective is to make an emergency brake to prevent a collision between a truck and an obstacle in front of it. If a collision cannot be prevented anymore, the system should also decelerate the truck thereby minimizing the impact of a collision.
The EBS system consists of a distance sensor which measures the distance to an obstacle in front of the truck. When the truck comes too close to the obstacle, the EBS can warn the driver and other road users by means of warning lights and acoustic signals. When the trucker does not brake after having been warned, the EBS will initiate an emergency braking.

The EBS is a safety critical system. The most obvious hazard is that the system initiates an emergency braking when this is not necessary. This may cause that the vehicle driving behind the truck bumps up against it. In the future, the upcoming safety standard ISO/WD 26262 will apply to safety critical systems like the EBS. The ISO/WD 26262 is still a working draft and does not apply to safety critical systems, yet. However, once the standard has been finished, systems have to comply to the standard.

The ISO/WD 26262 for the automotive industry introduces the concept of a Safety Case. This concept has been used for a number of years in safety standards that apply to the nuclear, defense, aerospace, and railway industry [27]. However, each safety standard of these different industries has a slightly different definition of the term Safety Case. This thesis defines a Safety Case as a document that communicates a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context. The exact meaning of the terms used in this definition can be found in Section 2.1.

Research has been performed that tried to find out how a Safety Case may be constructed [27]. This can be done by means of stating claims about a system, for example, “the EBS system is safe”. The trouble is to find evidence that supports this claim, just enumerating evidence does not make any sense while it lacks a coherent structure that makes clear what the evidence actually proves. A way to find supportive evidence is to decompose the top-level claim into sub claims; this is recursively repeated until claims are found which can be justified by evidence. The resulting claim structure usually will have the form of a tree, where the leaf claims will be connected to evidence.

As depicted in Figure 1.1, Weaver [49] proposes to decompose a top-level claim into at least three sub claims. The meaning of the symbols has been explained in Appendix A. If it is possible to prove that system safety requirements have been validated, satisfied, and are traceable, then the resulting system is considered to be acceptably safe. System safety requirements are high-level requirements that concern the safety of a system. An example is “The EBS system shall not initiate an unwarranted emergency brake”. The system safety requirements are usually refined in safety requirements which are more tangible. A more exhaustive explanation of the relation between system safety requirements and safety requirements, as well as an elucidation of the development process at DaimlerChrysler, can be found in Section 2.2.1.

During this Master’s project, the traceability branch of the tree has been studied in depth. Traceability means that it is possible to trace requirements during the development process. As stated by Brügge and Dutoit [6], a typical development process consists of the following phases: requirements engineering, design, imple-
1.3 Problem Statement

As stated in Section 1.2, the ISO/WD 26262 requires that a Safety Case is constructed for prospective safety related systems in the automotive industry. First of all, Chapter 2 will describe the concept of the Safety Case and of traceability by showing what has already been found out about these topics.
Second, a method has been defined that shows how a Safety Case can be generated automatically. A method consists of a product model and a process model [23]. The product model defines which documents are needed to generate the Safety Case. Therefore, an analysis should find out which data is needed to create a Safety Case. This model should also include a document that informs management about the safety status of a particular project. The process model should make clear how the documents from the product model are transformed to one another and how they will be used to generate the final Safety Case document.

Third, we developed a tool that automates the method described above. The tool should be able to generate a evidence for a Safety Case, given a data set with traceability associations and a Safety Argument. Furthermore, developers should be able to quickly navigate to the development data that need to be reworked.

Finally, an evaluation has to be made that makes clear how well the method and the tool perform. This will be done by means of a case study. Subject of this study is the EBS.

Summary of the identified research questions:

• (definition—Chapter 2) What is meant by a Safety Case, considering the context of the automotive industry?

• (method—Chapter 3) What does a method looks like that is able to generate a Safety Case. The method should define a:

  1. product model which describes which documents are needed to generate the Safety Case and what these documents look like.

  2. process model that describes how the various documents defined in the product model have to be processed in order to produce the Safety Case.

• (tool—Chapter 4) What does a tool look like that automates the generation of Safety Cases?

• (evaluation—Chapter 5) Do the method and tool meet their objectives?

1.4 Summary and Outline of This Thesis

Safety has been defined as the freedom from unacceptable risk. Legislators have created safety standards that describe how a safety related system has to be developed. An example of such a standard is the IEC 61508. Another upcoming standard for the automotive industry is the ISO/WD 26262. This standard is still a working draft and contemporary systems do not have to conform to the standard, yet.

The IEC 61508 and ISO/WD 26262 give best practices on how a manufacturer should produce the system. Furthermore, they prescribe which constraints apply to a newly created system. For example, the ISO/WD 26262 introduces the concept of a Safety Case. A Safety Case is a document that communicates a clear, comprehensive
and defensible argument that a system is acceptably safe to operate in a particular context. We refer to this argument as the Safety Argument.

Before a system can be sold, inspecting agencies should approve the safety of the system. This means that they will verify whether the system conforms to its standard. Chapter 2 will show what has already been written about Safety Cases in literature.

A possible way to specify a Safety Argument is by means of a tree structure. The nodes of the tree consists of claims, where each child claim should refine its parent. The leafs of the tree consist of evidence that justifies a claim. One branch of the tree should show that the system safety requirements are traceable. Traceability means that it should be possible to track the safety requirement over all stages of the development process. Chapter 2 will also provide information about a state of the art traceability approach at DaimlerChrysler.

What the Safety Argument looks like with respect to traceability, has been described in Chapter 3. Furthermore, a method will be introduced that shows how a evidence for the Safety Argument can be constructed.

Based on the development data and the information stored by ToolNet, it is possible to automatically evaluate the Safety Argument and generate evidence. The tool that accomplishes this will be described in Chapter 4.

How well the method and tool meet our objectives will be evaluated in the Chapter 5. Finally, the wrap up of this thesis will be given in Chapter 6.
Proving that a system is safe is not new. Building contractors in the nuclear industry have to prove that a nuclear power plant is safe. The same holds for the manufacturers of trains and those who produce products for the armed forces.

Demonstrating a system’s safe can be done by means of a Safety Case. The Safety Case provides an argument why a system is safe.

Currently, a new safety standard for the automotive industry is under construction. This safety standard states that a Safety Case should be created for systems in the automotive industry. Section 2.1 describes what has already been written about Safety Cases by safety experts. This information will be used in Chapter 3 to be able to describe a method to generate Safety Cases.

The Safety Argument presented in the Safety Case is different for every system, although there are some commonalities between them. What the syntax and semantics of a typical Safety Argument look like, has been described in Section 2.1.1.

A part of the Safety Argument should address traceability. Traceability means that the safety requirements for a system should be tracked during the development process. This ensures that all safety requirements have been implemented in the final system. Section 2.2 elaborates the development process of DaimlerChrysler and how the requirements are traced over such a process.

2.1 The Safety Case

To convince inspectors that a system is safe, a Safety Case should be created. The safety standard ISO/WD 26262 [38] prescribes that a Safety Case should be created for every safety related system. In this standard, the term Safety Case has been defined as “Part of the documentation proving the fulfillment of safety requirements and thereby guaranteeing functional safety”. However, the standard does not make clear which elements the proof should contain to guarantee functional safety. Furthermore, no strategy has been proposed how such a proof can be constructed. Luckily, Safety Cases have been part of other safety standards and research has been carried out that proposes a method and notation to create Safety Cases [27].
2.1 The Safety Case

Kelly [27] defines a Safety Case as follows “A Safety Case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context”. The following items from the definition are especially important

- **clear, comprehensive and defensible argument**: the argument should make clear that it is reasonable to accept that the system is acceptably safe.

- **system**: every safety related system can be described with a Safety Case. This thesis mainly concentrates on embedded safety related systems manufactured by DaimlerChrysler.

- **acceptably safe**: as stated in Chapter 1 absolute safety cannot be obtained in reasonable time. A system is said to be acceptably safe if every safety requirement fulfills its Safety Integrity Level [10]. A Safety Integrity Level (SIL) specifies how high the probability $P$ of a dangerous failure per hour may be. The SILs and their respective values of $P$ have been listed in Table 2.1.

Table 2.1: Safety Integrity Levels

<table>
<thead>
<tr>
<th>SIL</th>
<th>$P$ in a high demand or continuous mode of operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIL4</td>
<td>$10^{-9} \leq P \leq 10^{-8}$</td>
</tr>
<tr>
<td>SIL3</td>
<td>$10^{-8} \leq P \leq 10^{-7}$</td>
</tr>
<tr>
<td>SIL2</td>
<td>$10^{-7} \leq P \leq 10^{-6}$</td>
</tr>
<tr>
<td>SIL1</td>
<td>$10^{-6} \leq P \leq 10^{-3}$</td>
</tr>
</tbody>
</table>

Assume that the ABA has been developed according to SIL3, then the probability that the requirement “The ABA system shall not initiate an unwarranted emergency brake” will be violated should be between $10^{-9}$ and $10^{-8}$ per hour.

- **context**: a system is supposed to be safe if it operates in a particular context and it is unwise to use the system outside this context. Consider paving stones; even these can be dangerous for example when thrown to a police officer. For a safety related system, the context should be demarcated in which the system can be used safely.

Amann [5] describes which elements a Safety Case report should contain. The main elements consist of a description of the system, an overview of the safety standards and SILs that apply to the system. Furthermore, an outline should be given how safety is managed and the development process should be described. Finally, the Safety Argument should be presented.

Traditionally, Safety Case reports have been written on paper. As stated by Cockram and Lockwood [9], a Safety Case for large systems can easily fill library

10
shelves. To deal with this large amount of data, they propose to create an Electronic Safety Case. The advantage of an Electronic Safety Case over a Safety Case written on paper is that it is easier to navigate through the large amount of material. For instance, hyperlinks can be used to reference key documents (existing in computer memory) that contain evidence.

For the automatic evaluation of Safety Argument—i.e., that part of the Safety Case that relates evidence and a claim to one another—the Electronic Safety Case is important. This will be further explained in Chapter 3, but first, a state of the art way to construct a Safety Argument will be presented in Section 2.1.1.

2.1.1 Safety Argument

Kelly [27] defines a notation, the Goal Structuring Notation (GSN), with which a Safety Argument can be defined. The GSN is based on earlier work by Toulmin who describes how a generic argument can be specified [44]. Such a generic argument consists of claims whose truth should be proven. The facts that are used to prove the claim are referred to as data. The relation between claims and data corresponds to the logical expression \[ \text{data} \rightarrow \text{claim} \]. Why the data is a proof for the claim is described by warrants. When it is possible to dispute the warrant—because it is not always possible to give a formal proof—backing can be used that shows why the warrant is valid. How these argumentation elements are related, has been depicted in Figure 2.1. For a complete overview of the Toulmin’s argumentation, the reader is referred to [44].

The main elements of the GSN are goals and solutions (also termed evidence). A goal corresponds to a Toulmin claim, whereas solutions relate to Toulmin data. Amman [5] and Weaver [49] try to tackle the problem which solutions are needed for a particular Safety Argument and why (Toulmin warrants and backing) these solutions supports the goal. The idea is that a Safety Argument in the GSN starts with a top level goal, mostly this claim is “The system is safe”. This top level claim is recursively decomposed into sub claims until a sub claim has been found which can be easily proven by evidence. The warrant, in the GSN the structure represented
There exists a notation to depict the goals and solutions in a graphical way. This graphical notation represents goals and sub goals (respectively claims and sub claims) as rectangles and solutions as circles. During this thesis project a tool, which is called ASCE [32], has been used to create GSN networks. An example GSN network, which has been taken from the example Jaguar XK8 Electronic Throttle Safety Case [16], has been shown in Figure 2.2. Notice that only a part of the original Safety Argument has been depicted.

![Figure 2.2: An example definition of a Safety Argument specified in the GSN](image)

Figure 2.2 introduces some other symbols, being the strategy, the justification, the model, the assumption, and the context. These elements can be used to clarify the argument and are, therefore, comparable to warrants and backing. A strategy element is used to add details why a goal has been decomposed into sub goals. Sometimes, justification is needed (Toulmin backing) to show why a certain strategy can be applied. A model symbol can be used to reference design information.
When assumptions are made when a claim is stated, these assumptions should be mentioned explicitly in the GSN model. Finally, a context symbol is used to make clear in what context a claim, strategy, or solution is stated.

Kelly [27] defines a six step method to construct an argument in the GSN. The method’s steps have been listed in Table 2.2. This table shows that basic goals and strategies should be defined which should be decomposed and elaborated until basic solutions can be identified.

Table 2.2: The steps the GSN construction method

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identify goals to be supported</td>
</tr>
<tr>
<td>2</td>
<td>Define basis on which goals stated</td>
</tr>
<tr>
<td>3</td>
<td>Identify strategy to support goals</td>
</tr>
<tr>
<td>4</td>
<td>Define basis on which strategy stated</td>
</tr>
<tr>
<td>5</td>
<td>Elaborate strategy (therefore proceed to identify new goals and go back to Step 1 or proceed with Step 6)</td>
</tr>
<tr>
<td>6</td>
<td>Identify basic solution</td>
</tr>
</tbody>
</table>

Although it is difficult to give a detailed specification of the content of an argument, there are some argument structures that stay the same for most Safety Cases. The argument structures that can be reused for several Safety Cases are termed safety case patterns. Safety case patterns are similar to design patterns [18]. By reusing the patterns, Safety Cases can be constructed faster. Furthermore, confidence can be put in the pattern argument if the rationale of that argument has been proven. Besides patterns, AntiPatterns have been introduced [49]. An AntiPattern can be used to communicate weak and flawed Safety Arguments. This prevents that common argumentation mistakes will be made over and over again.

Amann [5] has proposed a GSN structure that serves as a basis for a tree decomposition of a top level goal. The suggested decomposition has been depicted in Figure 2.3. The process branch of the tree should make clear that the way in which the system has been produced, is safe. Most of the times there are safety standards that prescribe what a development process should look like. What a typical development process for DaimlerChrysler looks like, has been described in Section 2.2.1. However, describing that the system has been developed in a safe way does not guarantee that the actual system is indeed safe. The product branch of the tree should make clear that the system itself is safe.

Chapter 1 has already introduced a GSN model 1.1 created by Weaver that describes a possibility decomposition of the product branch in a validation, satisfaction, and traceability branch. This thesis concentrates on the branch of traceability. Traceability will be explained in Section 2.2.
2.2 Traceability

As shown in Figure 1.1, one branch of the Safety Argument should give a proof that the data produced during the development process is traceable. The definition for traceability used during this paper has been taken from [38] “The origin, realization and proof for a requirement are clearly described in the documentation”. A requirement is a condition or an ability that a system should fulfill. The origin of a requirement is a rationale why this requirement has been elicited for the newly developed system. The realization should show how the requirement is implemented in the final system. A proof for a requirement means that it should be proven that the requirement has an origin and has been realized. How requirements by DaimlerChrysler are realized and how traceability information is managed has been described in this section.

2.2.1 Realization of Requirements

Usually, large systems are developed according to a development process. Brügge and Dutoit [6] describe a development process for typical software systems. Although most systems used in vehicles are embedded systems rather than generic software systems which run on personal computers, the development process for embedded systems in the automotive industry usually looks the same. Embedded systems are hardware platforms which usually have limited processing capability and a small amount of memory compared to desktop computers. Software programs that run on such systems are written in programming languages that only contain a small subset of the constructs that can be used with generic high level programming languages like C++ or Java.

Most of the embedded systems that are developed by DaimlerChrysler are safety critical. This means that a malfunctioning system can hurt or kill people. As described in Chapter 1, such a system is subject to safety regulations. Those safety
standards describe how a safety critical system has to be developed.

A prescription of the IEC 61508 safety standard is that the safety related functions of such systems should be developed and realized separately from the functional functions of the system. The safety related functions are those functions that ensure the safety of the system; all remaining functions are considered to be functional functions. If it is impossible to separate the functional functions from the safety functions, the system as a whole is a safety critical system. The ABA system, for example, is not split into two distinct parts, so the whole system is seen as a safety related system and therefore the whole system is subject to the rules of the safety standards.

Figure 2.4 shows a V-model [46] for embedded safety critical systems at DaimlerChrysler. This model shows the process steps according to which a system will be developed. For the ABA system, there is a distinction made between functional requirements and safety requirements during the requirements specification phase. However, in the subsequent phases the functional and safety functions have been merged.

**Requirements Specification Phase**  The requirements specification phase consists of the definition of functional and safety requirements. First, the construction of the functional requirements will be sketched. Second, an outline of the safety requirements construction will be presented.

The construction of functional requirements starts with the definition of top level requirements. Top level requirements describe at a high level which requirements the final system should fulfill. An example top level requirement for the ABA system is: “A system should be created for Mercedes trucks that brakes automatically if an accident is about to happen”.

The top level requirements are refined in system requirements which means that the functions of the system are described at a detailed level. Then, the system is decomposed into several subsystems—also termed Electronic Control Units (ECUs)—and the high level system requirements are refined in various subsystem require-
2.2 Traceability

Example ECUs for the ABA system are the Brake, Distance Sensor, Radio, or Emergency Brake subsystem. An example subsystem requirement for the radio subsystem is: “The radio should be muted as long as a ‘mute’ signal is on the CAN bus”. The CAN bus [25] provides a communication channel which makes it possible for systems in the truck to send messages to or receive messages from one another. Muting the radio is necessary because the ABA system warns a driver by means of an acoustic signal. The radio should be muted to ensure that the truck driver hears the warning signal.

The component requirements are further split into hardware and software requirements. This thesis’s main focus is on software requirements. The level of detail of the software requirements should be sufficient to design the complete functional part of the system.

As said before, safety requirements should be specified, too. To identify safety requirements, hazards for the system need to be identified. This can be done by techniques like the Fault Tree Analysis (FTA) or Hazards and Operability Analysis (HAZOP) which have been described by [30]. When the hazards have been identified, safety goals are defined for each hazard. Safety goals can be compared to top level requirements while they also specify requirements at a high level. The safety goals are refined in safety requirements which are allocated to the various ECUs; for this reason, safety requirements can be compared to functional system requirements. The ABA system combines the functional and safety functions of the system. As a result, both safety requirements and functional system requirements have been implemented by software or hardware requirements. This implies that no distinction is made between the safety and functional functions of the system during the Design phase and Implementation phase.

Notice that it is often difficult in practice to uniquely assign a certain requirement to a requirement level. This is while the requirements might have a varying level of detail although they reside at the same requirement level. One developer may interpret a requirement as a high level software requirement, whereas another developer argues that the same requirement is a low level system requirement. More research is needed that proposes how to deal with this problem in the future.

Most embedded systems in the automotive industry are designed by making use of the Model-Based Development (MBD) paradigm described by Rau [40]. As the name implies, models are the most important artifacts that are used during the development process. The starting point is a simple model which will be further refined in the course of the development process until code can be generated out of it. A number of tools, described in Section 2.2.1.1, have been developed to support the MBD process. There exists, for example, a tool that make it possible to semiautomatic generating code out of the specified models.

**System Design Phase** During the system design phase, the system and software requirements are translated into a functional model of the system. The functional model consists of a architectural model and a component model. As soon as the
system requirements have been specified, an architectural model can be created for those subsystems that do not already exist. By creating an architectural model, some deficiencies might be identified in the system requirements after which such a particular requirement can be revised.

**Hardware and Software Design Phase** The hardware and software design phase specifies what the hardware and software looks like. The software requirements are transferred into the component model which is a refinement of the architectural model. Again, the software or hardware requirements may be reworked to correct ambiguous requirements. The architectural and component model are often referred to as the functional model. Notice that not only functional functions are present in the functional model but also safety functions.

**Implementation Phase** When the functional model has been defined, the model can be transferred in an implementation model. This is done during the implementation phase. The implementation model can be used to generate C code. This cannot be done completely automatically while the developer is needed to resolve ambiguous situations which the code generator cannot solve on its own.

**Integration and Assessment Phase** During the integration and assessment phase, the ECUs will be merged and the new system will be tested at various levels. It is possible to test the functional and implementation model at system and component level. These tests are termed Module in the Loop tests (MiLs) and are used to test if the model is valid. These tests are very useful because they show in an very early stage of the development process if the system can be implemented as specified in the design model. The code that will be generated from the implementation model also can be tested at system and component level. This test phase is called a Software in the Loop test (SiL). The SiL tests if the software conforms to its requirements. Furthermore, some static analyses, like code coverage, can be performed on the generated code.

When the SiL is successfully executed, the software can be integrated with the hardware and each ECU is tested in the Component Test Bench (CTB). Then, the various ECUs needed for the system are integrated and a Hardware in the Loop test (HiL) is performed on the new system. The following test is the VEHicle in the Loop test (VEHiL). Here, the system is tested in a non-moving vehicle to see if new system also functions in the environment where all systems are active. The final test is a trip of 50 km where the system is tested in its real world environment.

The assessment phase does not only consist of testing; it is necessary to check all development data elements for consistency, feasibility and completeness [12], this is also referred to as reviewing. Example issues that should be considered are “Does this requirement really cover the requested property” or “Is this code snippet really implementing a requirement”. At this moment, answering these questions cannot be done automatically. Therefore, reviewing has to be done manually.
As might be clear, a lot of data is generated during the development process. Section 2.2.1.1 lists tools that are used to create and manage this large amount of data.

### 2.2.1.1 Tools Used During a Development Process

Table 2.3 list tools that are used during each development phase of embedded systems at DaimlerChrysler. The tools can be used to easily and rapidly create data that is necessary for the development of a system. Furthermore, the tools are indispensable to manage the large amount of data that is produced during the development of a system.

Table 2.3: Tools used during the development process at DaimlerChrysler

<table>
<thead>
<tr>
<th>Design Phase</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Analysis</td>
<td>DOORS</td>
</tr>
<tr>
<td>Design</td>
<td>Matlab Simulink</td>
</tr>
<tr>
<td></td>
<td>Matlab Stateflow</td>
</tr>
<tr>
<td>Implementation</td>
<td>TargetLink</td>
</tr>
<tr>
<td>Assessment</td>
<td>Tessy</td>
</tr>
<tr>
<td></td>
<td>TPT</td>
</tr>
<tr>
<td></td>
<td>MTest</td>
</tr>
<tr>
<td></td>
<td>CTE</td>
</tr>
<tr>
<td></td>
<td>QAC</td>
</tr>
<tr>
<td></td>
<td>Polyspace</td>
</tr>
<tr>
<td></td>
<td>Quality Center</td>
</tr>
<tr>
<td>Various</td>
<td>ToolNet</td>
</tr>
</tbody>
</table>

To store the requirements, DOORS is used. This is a tool that can manage large amounts of requirements. The basic functionality is the creation of tables which can hold data elements. Not only functional requirements, but also safety requirements and review reports are stored in DOORS. As stated before, nearly all requirements are specified in natural language, but DOORS can also deal with images.

There are several tools that support MBD. The most important tools for creating design models are Simulink and Stateflow. Simulink is an extension to Matlab and can be used to make a model of the system. A model typically consists of building blocks with input and output signals. A mathematical function is used to map the input of the block to output. Stateflow is an extension to Matlab and Simulink. In Stateflow, the system is represented as a state machine. This representation can be used to simulate and analyze the system.

Semiautomatic code generation out of Simulink or Stateflow models can be done with TargetLink. The developer has to make the models ready for TargetLink, which
can take quite a large amount of time. Therefore, we do not claim that the code generation process is fully automatic.

Proving that a system is doing what it should do is a problem that can be reduced to the halting problem \[36\] and therefore it is NP-complete. To test the system in reasonable time, test cases have to be found that cover the system as much as possible. Determining test cases in a systematic way can be done with CTE (Classification Tree Editor). Once test cases have been specified, unit tests can be executed by using Tessy. For static analysis QAC and Polyspace can be used. TPT (Time Partitioning Testing) is used to simulate continuous signals that operate on the system. This program can be used to test models of the design stage. To manage test cases and there results, Quality Center is utilized.

The massive amount of data stored in the tools can be managed by ToolNet \[4\]. ToolNet can create associations between data elements stored in tools. Based on the information stored in ToolNet, an analysis can be executed that show whether or not the requirements have been realized. Section 2.2.2 describes the atomic units—termed Development Objects—between which an association can be drawn with ToolNet. Then, Section 2.2.3 explains how those associations can be created.

### 2.2.2 Development Objects

The development data created during the development process is stored in development tools. To be able to create associations between data elements, atomic units should be defined between which associations can be drawn. The atomic units are termed development objects. What is considered as development object is subjective. Take, for instance, a file containing a definition of a C++ class. Somebody can argue that a class is a development object. However, another person can define its functions as development objects and another person even can consider characters of the file as development objects. Furthermore, not all development objects are the same for every development project. For one project, C++ functions are development objects, but for another project it might be sufficient to define classes as a whole as development object.

To be able to use development objects in a tool and creating associations between them, the development objects used during a typical development process should be defined unambiguously. A way to do this is by means of a development object model which has been introduced in \[26\]. Such a model describes which development objects are available during a particular development process. This model is termed Information Model (IM) in ToolNet \[2\]. An example IM for a requirements specification phase, is given in Figure 2.5. The model has been specified in UML \[22\] whose class concept can be reused for the development objects. The elements specified in the model can be interpreted as development object classes. Various development object instances can exists for a given development object class. For instance, in DOORS several safety requirements at instance level can be created which all belong to the development object class SafetyRequirement.
2.2 Traceability  

The example model describes which development object classes for the requirements phase have been specified. The branch for the functional requirements consists of TopLevelRequirements and SystemRequirements. The branch for the safety requirements starts with Hazards, followed by SafetyConcepts and SafetyRequirements. Finally, both branches have been merged in SoftwareRequirements. It may be clear that this classification resembles the requirements process steps described in Section 2.2.1.

The only issue that is left is how to map a data element in a tool to a development object class. For the requirements this mapping is straightforward, for every development object class a table can be created in DOORS. Every data entity specified in the table belongs to the development object class which name equals the table’s name. The idea of defining development object classes and link data from tools to these classes cannot only be used for requirements, but also for design elements, tests, reviews, or any other kind of data element.

In ToolNet, the mapping between the data elements of the tools and the IM is done with a Tools For Projects Model [2]. An overview of the relations between the development tools, the IM and the Tools For Projects Model is given in Figure 2.6.

At the lowest level the development tools are depicted. The data elements stored in these tools are described with a model, the Tool Model [2]. For every tool used during the development process a Tool Model can be created. The model is usually static, which means that it does not change that often because the data that can be stored in the tool usually does not change, except when a new version of the tool is released. This means that a given Tool Model can be reused for every system
development process, even between various companies. The Tool Model defines the smallest possible development objects that can be referenced for a particular tool and thereby it also determines the granularity of the IM. Consider the example of the file with the C++ class, if the Tool Model only specifies entities at class level, then the IM cannot reference the class’s functions because these functions have not been modeled in the Tool Model.

The Tools For Projects Model defines a mapping between the various Tool Models and the IM. The mapping between the models can be done as follows. Every object from the tool model is inspected and mapped onto a development object from the IM. This mapping can be performed automatically by following some rules which are termed Type Hints. Figure 2.7 shows how a requirement (in German: Anforderung) from the IM can be mapped onto a FormalObject from the Tool Model. This happens with the help of a Type Hint which maps the FormalObject onto a Requirement by using the name of the module in which the FormalObject resides.

The ultimate goal of the traceability realization is to show that all development objects have been associated with one another. How these associations can be created has been described in the next section.

2.2.3 Creating Associations Between Development Objects

When the development objects have been uniquely defined, it is possible to create associations between them. This can be done with ToolNet [3]. By creating associations between development objects, it becomes possible to trace which objects are connected to one another. Figure 2.8 visualizes associations between development objects created during various development phases with manifold tools. All associations created with ToolNet do have a type. An association’s type should be specified in the IM. When the developer does not specify a type, the association will be linked to a generic predefined type. The type can be used for documentation.
2.2 Traceability

The development object instances in Figure 2.8 are represented as squares and the name of the class to which an instance belongs, is written to the left in the same color as the square. As can be seen, all nodes from the graph have incoming or outgoing links, which means that traceability can be proved for the depicted system. Notice that the graph depicted in Figure 2.8 only contains an example subset of development objects and is by no means meant to show associations between all development object instances created during a development process.

The graph visualized in Figure 2.8, depicts that a safety requirement which can be traced over the whole development process. The safety requirement has been associated with a review. The review describes if the requirement is relevant and has been formulated in a clear way. The graph shows furthermore, that the requirement has been split in two software requirements. Notice that the semantics of the development object instances are not relevant for traceability. Development object instances only are identified and associated with one another, without knowing which details the instance contains. Then, the software requirements are mapped onto design elements which are implementation in C-functions. Finally, the implementation is tested with unit tests. As state before, this graph is by no means complete, while for instance hazards or MiL tests have not been taken into account.

ToolNet can be used to create associations between development object instances. The graph depicted in Figure 2.8 could for example be created by ToolNet. This can be done by simply selecting two development objects. Then, the user can instruct ToolNet to create an association between these instances.
2.3 Summary

To prove that a system is safe, the safety standard ISO/WD 26262 prescribes that a Safety Case should be created. A Safety Case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context. The argument presented in the Safety Case provides evidence why a system is safe. Specifying a Safety Argument can be done with the Goal Structuring Notation (GSN). The main elements of this notation are goals, structures, and solutions. Goals are claims which are stated by a safety expert. These goals should be proved by evidence, called solutions. However, just listing some solutions for a particular goal is not sufficient. An argument should be given why a particular piece of evidence proves that a claim is valid.

One part of the Safety Argument should prove that all safety requirements have been implemented in the final system. This trace of safety requirements during the development process is called traceability. To be able to trace the requirements, the development objects that are created during a development process should be defined. Then, it is possible to create associations between these objects. DaimlerChrysler uses the tool ToolNet to define the development objects and create the associations between them.

This chapter is meant to provide general information about concepts that will be used during the next chapters. Chapter 3 defines what the traceability part of the Safety Argument looks like. Furthermore, it will describe how evidence for the Safety Argument can be generated.
The Safety Argument is an important part of the Safety Case because it gives an argumentation and evidence that shows that a system is acceptably safe. Evidence can be generated for some parts of the Safety Argument. We have defined a method to generate some part of the evidence. This method will be introduced in this chapter.

A method consists of a product model and a process model [7]. A product model describes which documents are relevant for a method. The process model describes how the documents defined in the product model do relate to one another and which steps are needed to transform a particular document into another one.

The Safety Analyzer and the ISO/WD 26262 postulate requirements to the output documents of the evaluation method. The requirements to the method will be described in Section 3.1. Based on these requirements, the product model will be described in Section 3.2 which defines, among others, how a part of the traceability Safety Argument can be constructed. Then, Section 3.3 will provide a description of the process model of the method.

3.1 Evaluating a Safety Case

Evaluation of the Safety Case has several purposes. First of all, the evaluation result should be used to inform managers about the system’s safety. This requirement comes from the Safety Adviser. Second, the results can be used to show that the system is acceptably safe. These results, written down in a report, is required to convince TÜV inspectors of the safety of the system. Notice that this thesis concentrates on the traceability branch of the Safety Argument and not about the overall safety of the system.

Informing managers about the system’s safety can be done by making use of the Safety Argument specified in GSN. The Safety Argument can be augmented with color information, in a similar way as for the Project Adviser, shown in Figure E.1.
3.1 Evaluating a Safety Case

The Colored Safety Argument should give a graphical overview that makes it possible for a manager to quickly identify which part of the solutions is incorrect. The incorrect solutions should be depicted with a red color whereas the correct ones should be green. The color information should ripple through the whole Safety Argument. This means that the top level goals will be colored red if at least one solution is incorrect, otherwise, the goal will be green.

Evidence for the Safety Argument should be generated in order to decide whether a branch of the Safety Argument is valid (green) or not (red). To generate evidence, the development objects and their associations should be evaluated. The solutions referenced in the Safety Argument should contain pointers to the development objects in the development tools which have or have not been associated to another development object so that developers can quickly navigate to the development objects that have to be reworked.

The generated evidence can also be used to in a report that will be handed over to safety inspectors. The Traceability Report should contain an overview of all development objects. Furthermore, the trace information to other objects should be listed. The traceability report is a part of the Safety Case report and can be used to convince safety inspectors that the safety requirements of the system described in the Safety Case are traceable.

The documents listed above are needed by a method that can evaluate a Safety Case and produce the desired documents for managers, developers and safety inspectors. Figure 3.1 presents a graphical overview of a method’s input and output.

![Figure 3.1: High level method overview](image)

The input of the method consists of development data, associations stored in ToolNet, and a Safety Argument. Chapter 2 has already introduced the development data, ToolNet, and a general notion on a Safety Argument.

The generation of the solutions—evidence—is of particular importance. Based on the evidence, a Colored Safety Argument can be created and developers will be able to quickly identify development objects that do not fulfill the traceability condition. Furthermore, the evidence forms the bases of the Traceability Report. The creation of solutions will be the most important topic of this chapter. The Colored Safety Argument and the Traceability Report are not subject of this thesis. More about these topics can be found in Appendix B.

The next section will describe the construction of the Safety Argument for traceability. Furthermore, an explanation will be given what the solutions look like which are referenced in a Safety Argument.
3.2 Product Model

The product model elucidates the documents that are used by the method that evaluates a Safety Case. Although the development artifacts have been described in Section 2.2.1, a short recapitulation has been given in Section 3.2.1. The development artifacts will be used to construct the Safety Argument. What the traceability Safety Argument looks like, will be subject of Section 3.2.2. Then, Section 3.2.3 will explain what the solutions look like, which are referenced in a traceability Safety Argument.

3.2.1 Development Data

The safety part of the development data consists of hazards, safety goals, safety requirements, software requirements, design elements, and implementation elements. Besides that, every development element has an associated review. Furthermore, test data at various levels is available. What is detailed description of the meaning of these data elements has been described in Section 2.2.1.

3.2.2 The Safety Argument

An initial layout of the Safety Argument has already been described in Section 2.1.1. We propose a modified version of the Safety Argument which has been depicted in Figure 3.2. This variant represents the traceability part of the Safety Argument.

As proposed by [5], the top level goal is split into a process and a product branch. The process branch of the Safety Argument will not be subject to this thesis. The product branch is further decomposed into a traceability sub goal which will be decomposed further with a traceability strategy. A complete Safety Argument typically has more product sub goals, but only the traceability sub goal has been listed here because this sub goal has been studied in this thesis.

As stated in Section 2.2, the ISO/WD 26262 defines that the origin, realization, and proof of the safety requirements should be documented. This is represented by the tree sub goals below the traceability strategy. Notice that this decomposition is slightly different from the decomposition given by Weaver [49] who places the safety requirements validation and satisfaction directly below the product goal. The validation and satisfaction sub goal give a proof for the safety requirements, therefore we decided to move these sub goals to the proof sub goal.

It turned out to be difficult to evaluate the proof sub goal, while validation and satisfaction cannot be computed automatically, but are reviewed by humans. The results of these reviews are stored in DOORS, but are not yet available in ToolNet. If these results were stored in ToolNet, the analysis of the results would be easier and could be done in a similar way as the evaluation of the traceability branch of the Safety Argument. We choose to concentrate on the origin and realization sub goal, because all trace information needed to evaluate these goals is accessible via ToolNet. In the future, it might be possible to access review results in ToolNet.
Then, the evaluation method described in this chapter can be easily extended and it will be possible to analyze the extra information. More information can be found in Appendix B.

Traceability can be checked in two directions, forward and backward. Forward and backward traceability is important, as illustrated by the definition of traceability given by Gotel [19] which states that “requirements traceability refers to the ability to describe and follow the life of a requirement in both a forwards and backwards direction”. Forward traceability checks if development objects from two consecutive development process steps, termed A and B, have been associated to one another. Backward traceability checks the reverse, i.e., if the elements from process step B are connected to elements from process step A.

Figure 3.3 visualizes forward and backward traceability. Development object classes have been represented as sets (ellipses) and development object element as set members (black squares). The first Figure 3.3a shows two sets, set A and B. Set A can represent, for example, the set of all hazards, and set B the set of all safety goals. Set A has four elements, of which two elements have been associated to elements from set B. The two other elements do not have associations to elements of...
set B. This means, for instance, that two hazards have not been transposed to safety goals, or, alternatively, the safety goals are there, but the associations are missing. Partial forward traceability means that not all elements from set A have associations to other sets. A similar argument can be given for full backward traceability. This means that all elements from set B have associations to members of set A.

Figure 3.3b shows full forward traceability, but partial backward traceability. This can be caused by the fact that needless safety goals have been defined. To make the set fully backward traceable, either the unnecessary members of set B have to be removed, or, associations to an element from set A should be created. The last Figure 3.3c shows two sets which are full forward and full backward traceable. The trace for these sets is said to be sufficient.

\[ \forall a \in A \rightarrow \exists b \in B \]  
(3.1)

\[ \exists a \in A \leftarrow \forall b \in B \]  
(3.2)

Forward and backward traceability conditions can be expressed with the help of predicate logic [43]. Equation 3.1 shows a general way to express forward traceability for elements from A to B; the antecedent of this implication is \( aA \), whereas \( bB \) is referred to as the consequence. The equation expresses that every element from A should have an association to an element in B.

Equation 3.2 shows a general way to express backward traceability for elements from B to A; the antecedent of this implication is \( bB \), and \( aA \) is termed the consequence. The backward equation expresses that every element from B should have at least one association to an element in A.

Once forward and backward traceability have been expressed in logical equations, these expressions can be evaluated and a valuation can be assigned to each equation. Assume, Equation 3.1 is applied to the set depicted in Figure 3.3a, then the valuation for this equation will be 0 (false) because there are some members from A that are not associated to an element from B. However, applying the same equation to Figure 3.3b the assigned valuation will be 1 (true), because every member of A has been associated with a member of B.

The forward and backward traceability can be used in the Safety Argument. The origin and realization sub goal of the Safety Argument, which has been depicted in
3.2 Product Model

A Method to Evaluate Safety Cases

Figure 3.2, can be further decomposed into a forward and a backward traceability goal. The decomposition of the origin sub goal has been described in Section 3.2.2.1, whereas the realization sub goal has been explained in Section 3.2.2.2.

3.2.2.1 The Origin of Safety Requirements

The origin of the safety requirements can be shown with backward traceability. As has been described in Section 2.2.1, the safety requirements stem from hazards and safety goals. All safety requirements identified for a particular system should be related to a safety goal. All safety goals, in turn, should be related to a hazard. If both claims have been assured, the claim that all safety requirements have an origin is true.

Although the need for forward traceability, i.e., verifying whether all hazards have a safety requirement, is not articulated by the safety standard ISO/WD 26262, we think it is essential for the safety of a system, because a hazard which has not been covered by a safety requirement may cause an accident in the final system.

The forward traceability sub goal can be decomposed further into two sub goals. The sub goals should represent that every hazard should have at least one safety goal. Furthermore, every safety goal should have a safety requirement. These claims can be easily expressed with predicate logic, see Equation 3.1, which is shown in Equation 3.3 and 3.4 where \( H \) is the set of Hazards, \( SG \) the set of Safety Goals, and \( SR \) the set of Safety Requirements. The backward traceability goal can be decomposed in the same way, see Equation 3.2. This can been seen in Equation 3.5 and 3.6.

\[
\forall hH \rightarrow \exists sgSG \tag{3.3}
\]
\[
\forall sgSG \rightarrow \exists srSR \tag{3.4}
\]
\[
\exists hH \leftarrow \forall sgSG \tag{3.5}
\]
\[
\exists sgSG \leftarrow \forall srSR \tag{3.6}
\]

Once the claims have been expressed in predicate logic, solutions can be provided for each equation. What a solution looks like, will be specified in detail in Section 3.2.3. For now, only a short example is listed. A solution should show that all elements of the antecedent have been associated to an object from consequence. A possible solution can be in the form of a trace table. For Equation 3.3, such a trace table should for each hazards \( h \) from \( H \), list the associated safety goal \( sg \) from \( SG \). When the hazard is not connected to a safety goal, an empty row will be shown in the trace table.

Figure 3.4 summarizes the decomposition of the origin goal of the Safety Argument. Notice that the ASCE tool used to create the figure does not have the ability to display quantifiers, therefore the quantifiers have been translated into natural language. The proposed decomposition of the origin goal can be used for every
new project that is initiated and should only be changed if a new state of the art development process will be proposed by safety engineers.

Notice, that it is, in principle, possible that a safety engineer defines a hazard and creates a direct association to a safety requirement. This is not advised, because this is a violation of the state of the art development process, which says that a hazard should be refined in a safety goal. It seems to make sense to define a sub goal in the Safety Argument that forbids a hazard to be directly associated with a safety requirement, or, in as equation \( \neg (\forall h H \rightarrow \exists sr SR) \). However, the Safety Argument does not specify such a goal. The reason is that the solution we choose to define development objects and their associations will automatically prevent this from happening. It is not possible to draw an association between development object elements if it has not been specified in the Safety Argument. More about this is written in Section 3.3.1.

### 3.2.2.2 The Realization of Safety Requirements

To show that the safety requirements have been realized, they should be tracked through all development stages. Section 2.2.1 already explained that safety requirements have been specified for every subsystem. The realization goal depicted in Figure 3.2 can be decomposed according to the Control System Architecture Break-
3.2 Product Model

A Method to Evaluate Safety Cases

down Argument pattern specified by Kelly [27]. This pattern specifies a strategy which decomposes a goal into sub goals, where each sub goal represents a sub system. Each of these sub goals can be further decomposed as wanted.

Equation 3.7 till 3.10 show the forward traceability steps for each subsystem. In these equations, the following abbreviations have been used, SR denotes the set of safety requirements, SWR the set of software requirements, DE the set of design elements, IE the set of implementation elements, and CE the set of code elements.

\[
\begin{align*}
\forall sr & \rightarrow \exists swr \quad (3.7) \\
\forall swr & \rightarrow \exists de \quad (3.8) \\
\forall de & \rightarrow \exists ie \quad (3.9) \\
\forall ie & \rightarrow \exists ce \quad (3.10) \\
\exists sr & \leftarrow \forall swr \quad (3.11) \\
\exists swr & \leftarrow \forall de \quad (3.12) \\
\exists de & \leftarrow \forall ie \quad (3.13) \\
\exists ie & \leftarrow \forall ce \quad (3.14)
\end{align*}
\]

The safety standard IEC 61508 [10] states that all development objects that do not have a function in the final system, should be removed. The rationale of this statement is that every development object can contain one or more faults, and thereby it can initiate unwanted, dangerous behavior of the system. A superfluous development object can be created when a developer implements a requirement which is later removed from the system or has been implemented by another developer in another function. With backward traceability, it becomes possible to verify if every development object has a predecessor, which implies that the development objects can be tracked down to the safety requirements, which means that the development object is necessary. The backward traceability steps have been listed in Equation 3.11 till 3.14.

A partial decomposition of the realized goal is shown in Figure 3.5. The Control System Architecture Breakdown Argument pattern has been applied and the argument shows two hypothetical subsystems. The subsystem goals can be decomposed further, but only the decomposition of subsystem 1 has been shown. The decomposition of subsystem 2 is the same as subsystem 1, but for the sake of brevity, the sub goals have not been displayed.

A subsystem goal is decomposed into a forward and backward traceability goal. The sub goals of the forward and backward traceability goal have been filled with Equation 3.7 till 3.10 and Equation 3.11 till 3.14, respectively. Again, the sub goals of the backwards traceability goal have not been displayed for briefness.

Evidence for the Equations 3.7 till 3.14 can be provided by means of trace tables. What these tables look like will be described in Section 3.2.3.
3.2 Product Model

3.2.2.3 About Tests and Reviews

The ISO/WD 26262 defines, among others, that the safety requirements should be realized. It has not been clearly defined that tests and reviews are part of the realization. However, as described in Section 2.2.1, tests and reviews form an integral part of the development process. During the development process, test specifications will be written for various test levels and the results of those tests will be written down in evaluation reports. Besides that, every development object should be reviewed. The question is, whether or not tests specifications and evaluation reports, as well as reviews, should be part of the origin or realization goal.

We think that test specifications, test results, and reviews, should not be a part of the origin and realization decomposition, but rather a part of the proof goals, shown in Figure 3.2. The reason is that test results and reviews are meant to verify the
content of development objects. Therefore, they rather contribute to the proof than
to the realization or origin of the safety requirements. The proof of the requirements
is not subject of this thesis. More information can be found in Appendix B.

3.2.3 Solutions

As Section 3.2.2 made clear, a part of the Safety Argument consists of solutions. The
solutions provide evidence for the claims stated in the argument. Goals that have
a solution as child are termed leaf goals, or leaf sub goal if the goal has a parent.
The equations of the leaf sub goals of the origin and realization goals all have the
pattern depicted in Equation 3.15. Notice that the backward traceability equations
also conform to this pattern which becomes evident when the pattern implication is
reversed.

∀aA −→ ∃bB

The pattern expression states that every element from the set A needs to map
onto at least one element in set B. The sets represent development objects classes,
for example the set of all hazards, or the set of all code elements. An element of a
set represents an instance of a development object class, so a particular development
object element stored in a development tool. Table 3.1 shows a generic trace table
that gives evidence for the generic pattern listed in Equation 3.15.

Table 3.1: Generic trace table for ∀aA −→ ∃bB

<table>
<thead>
<tr>
<th>a ID</th>
<th>b ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>an−1</td>
<td></td>
</tr>
<tr>
<td>an</td>
<td>bm</td>
</tr>
</tbody>
</table>

To be able to reference a development object element, such an element has
an identifier. The first column of a trace table will enumerate all the elements
of the antecedent (set A). Then, at least one ID of an associated element from the
consequent (set B) will be listed in the second column. In principle, this information
is enough to prove the claim. When every row of the second column contains an
element for every element in the first column, every element from the antecedent
has at least one associated element from the consequent, thereby proving that the
claim is true.

For humans, a list of IDs might not be that clarifying. Therefore, we have chosen
to list a short description for the listed IDs. Besides that, it is helpful when each row
of the element is automatically evaluated. This means that a new column has been
introduced which expresses with true or false if the claim is violated. Furthermore, a solution is said to be true if all elements from the evaluation column (column 3) are true. When one row shows a violation of the claim, then the whole solution is considered false. An example of the table with extended information has been given in Table 3.2.

<table>
<thead>
<tr>
<th>a ID</th>
<th>description</th>
<th>traceable</th>
<th>b ID</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>“description...”</td>
<td>true</td>
<td>$b_1$</td>
<td>“description...”</td>
</tr>
<tr>
<td>$a_2$</td>
<td>“description...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$a_{n-1}$</td>
<td>“description...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_n$</td>
<td>“description...”</td>
<td>true</td>
<td>$b_m$</td>
<td>“description...”</td>
</tr>
</tbody>
</table>

By means of Equation 3.3, which has been repeated below for convenience, an explanation will be given on what a typical solution for the pattern listed in Equation 3.15 looks like in practice. Remember that $H$ represents the set of hazards and $SG$ the set of safety goals.

\[ \forall hH \rightarrow \exists sgSG \]

Assume that, for a particular system, a developer has identified four hazards and refined them into two safety goals. Two of these hazards have been associated with a safety goal. A graphical representation of this situation can be found in Figure 3.3a where set $A$ represents the set of hazards ($HZ$) and set $B$ the set of safety goals ($SG$). To give evidence for this goal, a trace table can be constructed as shown in Figure 3.3. This trace table resembles the sample Table 3.2.

<table>
<thead>
<tr>
<th>Hazard ID</th>
<th>description</th>
<th>traceable</th>
<th>Safety Goal ID</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H8</td>
<td>“Das System...”</td>
<td>true</td>
<td>SG3</td>
<td>“In der...”</td>
</tr>
<tr>
<td>H3</td>
<td>“Durch Hinzunahme...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H9</td>
<td>“Wenn nachfolgend...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1</td>
<td>“Dem Missbrauch...”</td>
<td>true</td>
<td>SG9</td>
<td>“Auch wenn...”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a ID</th>
<th>description</th>
<th>traceable</th>
<th>b ID</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H8</td>
<td>“Das System...”</td>
<td>true</td>
<td></td>
<td>“In der...”</td>
</tr>
<tr>
<td>H3</td>
<td>“Durch Hinzunahme...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H9</td>
<td>“Wenn nachfolgend...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1</td>
<td>“Dem Missbrauch...”</td>
<td>true</td>
<td></td>
<td>“Auch wenn...”</td>
</tr>
</tbody>
</table>

Notice that the numbers in the IDs of the development objects do not have to be in successive order. This is while it is possible that during the development process,
development objects are deleted. This will not cause any problem, as long as the IDs are unique.

The example solution is false while there are two entries in the trace table that evaluate to false. It is even possible to color the rows that violate the claim so it is easier for developers to recognize the elements that lack associations to a safety goal. Furthermore, the table can be used as basis for further analysis. For example, an analysis can be performed on the table that filters only those rows which violate the constraint. The data from the trace tables can be used to create a Colored Safety Argument. This has been explained in Section E.2.

### 3.3 Process Model

The process model should show how the solutions can be constructed out of the Safety Argument, the development objects stored in the various tools, and the associations between the development objects. To do that, a process model, denoting the flow of information between the various artifacts, has been shown in Figure 3.6.

![Figure 3.6: Process steps needed to evaluate a Safety Argument automatically](image)

The analyzer component should transform the input to the output, so it should be able to generate solutions in the form of trace tables based on the Safety Argument and the development data and the associations between them.

As pointed out by [5], the safety development process should be defined during the construction of the Safety Case. Safety experts should know what the development process for the system described in the Safety Case looks like and which data elements are available. Based on this information, the Safety Argument can be constructed. This has to be done by decomposing the top level goal into sub goals. Then, the leaf sub goals of the argument can be constructed, e.g., by using the Equations 3.3 till 3.14. Finally, solutions of for the leaf sub goals should be added to the Safety Argument.

Using development objects in the Safety Argument does not make sense when it is not clearly defined what is meant by the development objects. Besides a definition of each development object, a generic way to access them is desired to perform analyses. The problem of uniformly access development data scattered over various tools has been known since long and several solutions to solve this have been proposed [4]. One tool that solves this problem is ToolNet. ToolNet defines development objects by...
means of models and the development objects are accessible (with some restrictions) via an interface.

To initialize the ToolNet models, information from the Safety Argument is needed. How the models can be constructed is subject to Section 3.3.1. Then, Section 3.3.2 shows how evidence can be generated from the solution symbols of the Safety Argument with the help of ToolNet and the data stored in the development tools.

3.3.1 Creating ToolNet Models

As stated in Section 2.2.2, ToolNet defines three important models, the Project Information Model (PIM), the Tools For Projects Model, and the Tool Model. The PIM is used to define the development objects that are available for a particular development process, whereas the Tool Model represents the data from the tools. Finally, the Tools For Projects Model defines a mapping between the PIM and the Tool Model.

The goals and solutions from the traceability part of the Safety Argument reference development object classes and specify which associations should exist between them. Example Equation 3.3, which has been repeated below, shows such an association specification from the Safety Argument.

$$\forall h \in H \rightarrow \exists s \in SG$$

Here, the relation between two development object classes has been defined. The elements from the set $H$ should be associated to at least one element from the set $SG$. When these classes are represented in the PIM, $H$ can be mapped with the Tools For Projects Model to Hazards and $SG$ to the SafetyGoals. The issue is how to create models out of this information. This will be explained in this section.

### PIM

In principle, creating PIM model elements from the Safety Argument can be done by creating two classes for every leaf goal. Both antecedent and consequent from Equation 3.3 till 3.14 should be represented in a class. Because the abbreviated names have been used in the equations, we think it is better to use the expressions stated in the solutions. This is no problem because the information from the leaf goals is the same as in the solutions.

A solution from the Safety Argument states that a “Trace table Hazard against SafetyGoal” should be created. Two classes should be generated for this sub goal and its solution, one for the antecedent and one for the consequent, being Hazard and SafetyGoal respectively. A graphical depiction has been shown in Figure 3.7a. This figure shows what the PIM looks like in the first step.

**TODO: GIVE THE ASSOCIATIONS A NAME!!!**

When classes have been created, it is not clear anymore which class represents the antecedent and which class represents the consequence. The relation between
3.3 Process Model

(a) Creating classes for the antecedent and consequence
(b) Creating directed associations between the development object classes
(c) Example PIM containing three object classes and two associations

Figure 3.7: Create a PIM

the two classes has vanished. To overcome this, an association between the classes can be created as has been shown in Figure 3.7b. The association is directed and points in the same direction as the implication, from antecedent to consequence.

Usually, the backward declarations do reference the same development objects as the forward traceability declarations. A check should be done in order to see if a certain class already has been added to the PIM. When a relation has been defined between the classes, no new association should be added. This implies that all classes from forward declarations should be added, followed by adding the classes for the backward equations. This ensures that the order of process steps is preserved in the model.

By iterating over all nodes of the traceability Safety Argument, classes and associations can be added to the PIM. This will result in a complete PIM when the whole traceability Safety Argument has been parsed. At this moment, the process steps have to be executed manually. However, a PIM can be created out of a Safety Argument automatically. This is a typical development model to model transformation problem which can, for example, be solved with the help of Triple Graph Grammars [28]. These Triple Graph Grammars transform an input model to an output model by means of a set of transformation rules. However, this has not been studied into detail during this thesis, see also Section 6.3 about this topic.

As stated in Section 3.2.2, the Safety Argument does not have to specify a safety sub goal $\neg(\forall h H \rightarrow \exists sr SR)$. This is while the PIM does not specify this association, therefore, this association cannot be drawn between development objects, and, therefore, this sub goal cannot be violated.

**Tool Model and Tools For Projects Model** The Tool Model of ToolNet is in principle not modifiable by end users. Every tool that is supported by ToolNet already has an accompanying Tool Model which defines the smallest possible elements that can be referenced in the PIM. This means that developers which create the Safety Argument should be aware of the smallest possible elements they can reference.
Assume that hazards have been defined in DOORS. All artifacts stored in DOORS are stored in modules. Each of the artifacts in the module can be references. For hazards, this means that every single hazard can be referenced. However, a hazard usually does not only have a description, but also some other attributes. At the moment of writing, these attributes cannot be referenced. More information is provided in Appendix B.

To map a development object from the PIM to a Tool Model, the Tools For Projects Model is used. This model defines which element from the Tool Model corresponds to the development objects defined in the PIM. The Tools For Projects Model should be edited until it contains all elements from the PIM. Pintat [39] describes how a PIM, a Tools For Projects Model, or even a Tool Model can be edited.

### 3.3.2 Creating Solutions

Once the models in ToolNet have been initialized and the development data has been added in the development tools, all information that is needed to create solutions is available. To generate evidence for a traceability sub goal, the solution for a sub goal can be parsed and the development objects can be extracted from the solution. Then, a trace table should be created that lists the development objects against each other. All development object elements of the antecedent can be requested from the corresponding development tool and for each element, the associations to the consequent can be requested and the consequence can be listed. When there is no associated consequent, the evaluation for that particular antecedent element will be false. Therefore, the whole equation is false. When all antecedent elements have a corresponding entry in the trace table, the solution evaluates to true.

The generation of solution shows a high level of commonalities. The same routine should be followed for the forward and backward traceability solutions. Therefore, a tool has been developed that automates the process of creating solutions. The tool will be explained in Chapter 4.

### 3.4 Summary

A Safety Argument should be constructed to prove that a system is acceptably safe. Part of the Safety Argument should prove that the safety requirements of a system are traceable. Being able to trace the safety requirements means that a proof of the safety requirements should be given, the origin should be documented, and the realization of the safety requirements should be shown. This chapter concentrates on documenting the origin and showing the realization of the safety requirements.

Showing that every safety requirement has an origin, means that every safety requirement should be connected to a hazard via a safety goal. Proving that the safety requirements have been realized means that the safety requirements have been realized during all development phases, thereby ensuring that every safety requirement has been implemented in the final system.
The Safety Argument should be enriched with a traceability argument that expresses that the safety requirements should have an origin and a realization. Once the traceability claims and solutions have been added to the argument, trace evidence should be generated. The evidence consists of trace tables which show the origin of every safety requirement. Furthermore, other trace tables should show that all safety requirements have been considered during every development process stage.

To fill the trace tables with data, the development data elements stored in the development tools should be accessible. A way to access this information is with the help of ToolNet. However, ToolNet’s models should be initialized first. These define which development objects are available. Furthermore, the trace links between the development objects have to be modeled in ToolNet. All information needed to initialize ToolNet’s models can be taken from the traceability Safety Argument.

Finally, the trace tables, which provide evidence for the claims of the Safety Argument, should be generated automatically, while filling the tables by hand is time-consuming and boring. Chapter 4 introduces a tool that is able to generate the trace tables automatically.
Chapter 4

A Tool to Generate Solutions

Once the Safety Argument has been created, it becomes possible to evaluate the argument. Evaluating the argument means that should be determined what the safety state of a system is with respect to the Safety Argument. To do so, the solutions which have been referenced in the Safety Argument should be analyzed. This can be done by generating trace tables for every solution. To determine whether or not the solutions and claims are true, the information in the trace tables can be used. Chapter 3 introduced a method that showed how the trace tables can be generated based on the development data, their relations, and the Safety Argument.

Without a tool, the generation of solutions for the traceability part of the Safety Argument would take a considerable amount of time while even the development data of a moderate system usually comprises over one hundred gigabytes of data. Creating tables based on such a large amount of information is time consuming and developers do not want to spend valuable time doing this.

We created a tool that is able to automatically generate trace tables based on development data, associations, and the Safety Argument. The tool is named Solution Generator, and will be presented in this chapter. The Solution Generator is a prototypical implementation, but it shows that automatic evaluation of traceability part of the Safety Argument is possible.

The requirements for the Solution Generator will be listed in Section 4.1. Then, Section 4.2 will present a design which shows how the prototype can be realized. Section 4.3 provides an overview of the implementation and integration of the design. Finally, Section 4.4 will summarize this chapter.

4.1 Requirements

As its name implies, the Solution Generator should generate solutions. The input for the tool consists of already existing components, namely the Safety Argument, ToolNet and the development data. As described in Section 3.2.3, the content of the solutions already has been defined. It is not necessary to represent the content
in the form of the tables, but all content that has been defined earlier should be available.

Besides the content of the trace tables and its representation, it should be possible for developers to quickly identify development objects that violate the constraints stated in the Safety Argument. Furthermore, it should be possible to navigate to the development objects that violate the constraints listed in the trace table. This means that it should be possible to jump to a development object by clicking on it.

As listed in Table 2.3, quite a number of tools are used during the development process of embedded systems at DaimlerChrysler. It should be forestalled that “yet another tool” will be introduced. There usually is some resistance against the introduction of new tools because developers already have to manage a lot of them. To prevent that a new tool is needed, existing tools should be used as much as possible. This prevents that developers have to learn a completely new tool. Furthermore, it is likely that the existing tools already provide functionality that can be reused. This will result in a less expensive product.

As stated in Chapter 3, only two patterns have been used in the traceability part of the Safety Argument and Appendix B defines some more patterns. However, a complete Safety Argument may define many more patterns. The Solution Generator should be able to deal with these patterns. Therefore, it should be easy to modify the tool.

Table 4.1 gives a summary of the requirements identified in this section.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sg_req1</td>
<td>The Solution Generator should generate trace tables</td>
</tr>
<tr>
<td>sg_req2</td>
<td>The input for the Solution Generator is the Safety Argument, associations from ToolNet, and development data</td>
</tr>
<tr>
<td>sg_req3</td>
<td>The content of the generated trace tables should be the same as the content specified in Section 3.2.3</td>
</tr>
<tr>
<td>sg_req4</td>
<td>No new tool should be created, but existing tools should be reused and extended</td>
</tr>
<tr>
<td>sg_req5</td>
<td>It should be possible to easily identify development objects that violate the constraint expressed in the Safety Argument</td>
</tr>
<tr>
<td>sg_req6</td>
<td>It should be possible to jump to a development object in a tool by clicking on a development object in the trace table</td>
</tr>
<tr>
<td>sg_req7</td>
<td>The Solution Generator should be extensible so it can deal with a wide variety of patterns which may be used in leaf goals</td>
</tr>
</tbody>
</table>

All requirements have an associated ID. This makes it easier to reference the requirements during this thesis.
4.2 Design

To fulfill the requirements the Solution Generator has been decomposed into several subsystems. A high level overview of the Solution Generator’s environment has been shown Figure 4.1. Here, the already existing components have been colored gray. The blue rectangle represents the Solution Generator, which will be subject of this chapter.

![Diagram of Environment of the Solution Generator](image)

Figure 4.1: Environment of the Solution Generator

The Safety Argument box represents the tool with which the traceability argument can be created. The chosen tool should export the Safety Argument to a format that can be easily read by other tools. This is necessary to be able to extract the logical expressions for the argument leaf goals and content of the solutions which have to be processed further by the Solution Generator.

The Solution Generator depends on ToolNet while ToolNet is specifically meant to store trace information. This information is needed to solve the traceability part of the Safety Argument. Furthermore, ToolNet has a function that can be used to navigate to a certain development object in a tool based on the ID of that object. This will be useful for developers to navigate to development objects listed in the trace tables.

The Development Data box abstracts from the various development tools. The box has been connected with ToolNet because ToolNet provides an interface to access the development data in the tools. This means that the development objects stored in the tools can be accessed in a uniform way. Hence, the data needed to fill the trace tables can be gathered in a standardized way.

When the Safety Argument has been constructed, the IM and Tools For Project Model (T4PM) can be constructed as has been explained in Section 3.3.1. For now, the construction of the ToolNet models has to be done manually.

4.2.1 A Technique to Create Solutions

The Solution Generator should create trace tables with the help of ToolNet’s data and the relations defined in the leaf goals and solutions of the Safety Argument. An example trace table has been depicted in Table 4.2 (which is a modified version of Table 3.3).
4.2 Design

Table 4.2: Trace table for $\forall hH \rightarrow \exists sgSG$

<table>
<thead>
<tr>
<th>Hazard ID</th>
<th>description</th>
<th>traceable</th>
<th>Safety Goal ID</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H8</td>
<td>“Das System...”</td>
<td>true</td>
<td>SG3</td>
<td>“In der...”</td>
</tr>
<tr>
<td>H3</td>
<td>“Durch Hinzunahme...”</td>
<td>false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two conditions have to be fulfilled to create trace tables. First, it should be known which development objects have to be validated. Second, those development objects from the various development tools and the associations between them should be analyzed. Then, the results of the analysis should be represented in the trace table.

To get the development objects whose associations should be validated, the traceability part of the Safety Argument should be parsed and the development objects should be extracted, for example in the form of equations, like $\forall hH \rightarrow \exists sgSG$. Another requirement to the Solution Generator is that it should be modifiable. It should be possible to define new patterns that do not look like the patterns specified in Equations 3.1 and 3.2.

Once the Safety Argument has been parsed and the logical expressions have been extracted, the development objects that will be subject to the analysis can be identified. First, all development objects for the antecedent of a logical expression should be gathered from the tools. Second, all associations for every object have to be validated. It has to be verified whether the object has been connected to at least one element from the consequent. Remember that it is not possible in ToolNet to create associations which have not been specified in the IM, so there is no need to check for invalid links.

Various techniques can be used to validate all associations for the antecedent. We identified three possible techniques, being SQL, Prolog, and OCL. A description of each of these techniques will be given in the remainder of this section. Furthermore, advantages and disadvantages of the techniques will be listed, followed by an explanation of the technique that seems to fit best with the requirements. Then, based on the selected technique, a design of the Solution Generator will be given. The various techniques make use of ToolNet functionality to evaluate the associations between development objects. An overview of ToolNet functionality will be given in the next paragraph.

**How ToolNet has Been Implemented** ToolNet provides a backbone where components can be connected to. One type of components are the ToolNet Enabled Tools, e.g., DOORS or TargetLink. A ToolNet Enabled Tool should implement some predefined ToolNet interfaces. By implementing these interfaces the tools provide functionality on which other components rely. For each ToolNet Enabled
Tool, a Tool Model should be created. Another type of component is the so-called Integration Component. Components of this type provide functionality to handle the development objects and their associations. A third ToolNet component type is the Visualization Component. This component is meant to visualize data stored in ToolNet. A graphical decomposition of a part of the ToolNet architecture has been shown in Figure 4.2.

![Figure 4.2: Architecture of ToolNet](image)

The ToolNet Enabled Tools are represented as diamonds, the Integration Components have a rectangular shape, and Visualization Components are depicted as hexagons. Please notice that only two ToolNet Enabled Tools are shown, but basically all tools from Table 2.3 can act as ToolNet Enabled Tool. The most important ToolNet components referenced in this thesis are the Relation Repository, the ToolProxy, and the Navigation Client.

The Relation Repository is meant to store the associations between the development objects. It is a relational database with a hibernate interface [1]. Hibernate provides an object oriented interface to the Relation Repository. Hibernate makes it easy to store objects which is of great use while ToolNet has been implemented with the object oriented programming language Java [29].

The development data stored in the development tools is only available to ToolNet when a development tool has been opened. However, the development data stored in that tool is not available anymore when a tool is closed. To make the development data accessible for analysis when a tool has been closed, the data from the tool should be stored in a database. This is the task of the ToolProxy [17]. The ToolProxy requires that a tool can export its data into an XMI format [21]. This format can easily be stored in an object oriented database called Meta Data Repository C.4. The ToolProxy is not available in the current version of ToolNet, but the functionality of the ToolProxy with respect to the export of development data and the Meta Data Repository (MDR) can be reused. Appendix C provides more details about XMI and the MDR.

The objective of the Navigation Client is to represent existing associations between development items. This is done by showing a list of development objects, their links, and their associated development objects. Please notice that every as-
4.2 Design

A Tool to Generate Solutions

Sociation stored in ToolNet is bipolar. This means that every link connects two development objects, one being the source, the other the target. Either the source or the target can reference a NULL object.

**Prolog**

It seems to be a good idea to use a programming language that is able to deal with logical expressions. After all, logical expressions have been used in the Safety Argument. Prolog is such a declarative programming language based on first order predicate calculus [43]. There is a free Prolog tool available termed SWI-Prolog [51]. The tool can be used to evaluate development objects with the help of logical equations.

To analyze the associations between development objects, there should be a way to access the development objects. SWI-Prolog can access development data that has been stored in a file or a relational database. However, such a file or database is not available in the current version of ToolNet. It is possible to create such a file or database by using the data exports which are also used for the ToolProxy.

To validate the data in the database against the expressions defined in the Safety Argument, the expressions only need to be changed slightly. Two Prolog expressions should be created for the pattern \( \forall a A \rightarrow \exists b B \), one expression should list all members of \( A \) that do have a reference to a member in \( B \), the second one should list all members that violate this pattern.

SWI-Prolog comes with a package that provides a bidirectional Java/Prolog interface. This implies that the result of both Prolog queries can be transferred to ToolNet’s Java environment. There, the results can be processed further and trace table can be generated. An example Prolog evaluation of a pattern will be shown in Section C.2.

**Structured Query Language (SQL)**

An alternative to Prolog is the SQL [15]. The SQL is a programming language that is used to query relational databases. The result of such a query is a new table containing the result. While the trace tables resemble tables from a database, the idea arose to generate these tables by means of the SQL.

To generate trace tables with SQL, the logical expressions from the Safety Argument have to be converted to a SQL expression, or, as an alternative, the Safety Argument is adjusted so it references SQL statements instead of logical equations. However, we think that it is more intuitive and concise to use logical expressions.

A property of the SQL is that it queries a database. However, such a database is not available in ToolNet. ToolNet specifies the development objects and the relation between them in the IM, but a general integrated access to this data does not exist. This is the same problem as discussed in the paragraph about Prolog.

The result of a SQL query is again a table. This table already has the form of a trace table. Section C.3 gives an example of the creation of trace tables with data from a hypothetical relational database. This example will show that it is indeed possible to generate trace tables with the SQL when the necessary data is available.
Object Constraint Language (OCL) A third alternative to validate the development objects and associations against the expressions defined in the Safety Argument is the OCL [48]. Originally, the OCL was meant to express constraints to UML models. Nowadays, it can also be used to define queries and reference values in those models. The OCL is an interesting technique while the ToolNet’s IM is a UML model, too. Therefore, the OCL can be used to specify constraints to the development objects and there associations modeled by the IM.

To specify OCL expressions, the logical equations from the Safety Argument should be converted to OCL expressions. This can be done by specifying a OCL expression for every pattern of logical expressions that has been used in the Safety Argument.

The IM is a model that stores the layout of the development objects. The actual development object instances have to conform to this model. As described in the paragraphs about Prolog and the SQL, no integrated view on the development objects and their links is available. For the OCL, the MDR can serve as a database which provides such an integrated view.

An OCL engine can be used to generate content for the trace tables. The results of the OCL engine will be in the form of boolean values indicating whether or not a certain constraint has been violated. OCL engines are available for Java, so integration of the engine with Java should not be a problem.

The Chosen Technique The previous paragraphs shortly introduced various techniques to validate development data and associations against the Safety Argument. Metrics to select a suitable technique can be distilled from the requirements.

1. degree to which the Solution Generator remains extensible
2. easy integration of the technique into existing tools
3. ease of use of the technique for the developers
4. time to market of the Solution Generator

The extendibility is extensible for all techniques. When a new pattern has been introduced in the Safety Argument, the pattern should be translated to the respective technique. Then, the new pattern immediately can be used in the validation process.

All techniques can be integrated into ToolNet. By making use of ToolNet, at least the functionality of the Navigation Client can be reused to present the trace tables. While ToolNet has been developed with Java and therefore makes use of objects, OCL is the best choice while it is based on defining constraints to objects. A Prolog or SQL solution would make use of relational databases which causes a discrepancy in technology making it more difficult to combine those techniques with ToolNet.
4.2 Design

OCL fits perfectly in the object oriented paradigm. This will make it easy for developers to use this technique. For example, specialists who have to construct the IM have to understand UML and will be familiar with object oriented programming. For those developers, it will be easier to use OCL than Prolog or SQL. Furthermore, ToolNet developers will program in Java and will also be familiar with the object oriented paradigm.

The time to market will be the shortest for the OCL. This is while the integrated view on the development objects and the associations can be stored in an MDR. Code from the ToolProxy can be reused to create the MDR. For both the SQL and Prolog, the database with an integrated view has to be created from scratch.

All in all, OCL seems to be the most promising technique to generate the trace tables.

4.2.2 The Design Made With OCL

Figure 4.2 already depicted input and output for the Solution Generator. Part of the input is from the Safety Argument which provides the Solution Generator with logical expressions. These statements should be converted to OCL expressions. The other part of the input consists of an integrated view of the development objects and the associations between them. Figure 4.3 shows a decomposition of the Solution Generator.

![Component decomposition of the Solution Generator](image)

Figure 4.3: Component decomposition of the Solution Generator

The Safety Argument should be read and parsed by the SAtoOCL subcomponent. The objective is to read the logical expressions and convert them to OCL expressions. Assume that the following expression has been used in a leaf goal of an argument $\forall h H \rightarrow \exists s g S G$, then the SAtoOCL component should recognize this expression.
as being an instance of the $\forall aA \rightarrow \exists bB$ pattern. In a lookup Table 4.3 should be specified by developers which maps logical expression pattern to an OCL expression.

Table 4.3: Lookup table from patterns specified in the Safety Argument to OCL expressions

<table>
<thead>
<tr>
<th>Safety Argument Pattern</th>
<th>OCL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall aA \rightarrow \exists bB$</td>
<td>context A inv: self.b-&gt;size() &gt;= 1</td>
</tr>
</tbody>
</table>

Then, the generic variables referenced in the expression can be substituted by the names from the solution linked to the logical expression. An example is shown in Listing 4.1. The context field denotes the class for which the constraint holds. For the example, this is the Hazard class. The second line specifies an invariant (inv). This denotes that the condition should always hold. The actual constraint specifies that self should have at least one attribute with the name safetyGoal. The example constraint has been specified for the IM depicted in Figure 3.7b.

Listing 4.1: OCL expression

```oclass
context Hazard
inv: self.safetyGoal->size() >= 1
```

The OCL expression can be used to validate the data and its associations. To process the data, the data and associations are stored in an object oriented database termed MDR C.4. This is done by the IM MDR component. Then, the OCL engine validates the data stored in the IM MDR against the previously defined OCL expression.

Finally, the results of the evaluation need to be presented in trace tables. To represent the data in a trace table, the Navigation Client of ToolNet will be reused because this ToolNet Integration Component can already display relations between development objects. Therefore, some minor changed to the Navigation Client will be necessary.

4.3 Integration and Implementation

The Solution Generator has been integrated with ToolNet while most of its input data comes from ToolNet. Furthermore, the trace tables are represented with ToolNet. However, the Solution Generator is neither an Integration Component nor a ToolNet Enabled Tool nor a Visualization Component.

The Solution Generator is not a ToolNet Enabled Tool while it does not contain development data. It rather uses the data from other ToolNet Enabled Tools. To be an Integration Component, the Solution Generator should provide some integration
4.3 Integration and Implementation

A Tool to Generate Solutions

functionality like the Relation Repository or the ToolProxy. However, the Solution Generator is an analyzer and not an integrator. Furthermore, it is obvious that the Solution Generator is not a Visualization Component while it delegates the representation of the trace tables to the Navigation Client.

While the Solution Generator does not fit in one of the existing ToolNet type categories, we name the new ToolNet component *Analysis Component*. It has been depicted as a circle in the ToolNet architecture in Figure 4.4.

![Figure 4.4: The Solution Generator in the ToolNet environment](image)

As has been depicted in Figure 4.3, the Solution Generator consists of four components, the SAtoOCL, IM MDR, OCL Engine, and the Visualization Preprocessor. The SAtoOCL component, responsible for the conversion of logical equations into OCL expressions, has not been implemented. Instead, the transformation has to be done manually. This transformation is the same as the one depicted in Listing 4.1, so the manual conversion should not be a problem.

The IM MDR is created by loading the IM into the MDR. Then, the Solution Generator gets all development objects from ToolNet and instantiates them in the IM MDR. The same has been done for all associations which are available in the Relation Repository. An algorithm in pseudo code to fill the IM MDR is given by Algorithm 4.1.

The OCL engine has not been implemented either, but has been taken as off-the-shelf component. Two OCL engines have been studied, an engine from the Technische Universität Berlin [8] and the Dresden OCL Toolkit [33]. It turned out that the OCL engine from the TU Berlin could be reused most easily while only a reference to the IM MDR and a reference to the file containing OCL constraints had to be passed to the engine.

Finally, the results from the evaluation returned by the OCL engine, need to be presented. ToolNet’s Navigation Client can be reused to present the results, an example of which is given in Figure 4.5. The evaluation results returned by the OCL engine cannot be interpreted by the Navigation Client. Therefore, the Visualization Preprocessor processes the results from the OCL engine and transforms them into a format that can be interpreted by the Navigation Client.

The screenshot of the Navigation Client in Figure 4.5 shows a part of an example
Algorithm 4.1 Pseudo algorithm to create a IM MDR

Require: All development elements are available in ToolNet
repeat
  Get development element from ToolNet
  Get the IM class the element belongs to
  Instantiate a IM element with the given type in the MDR
  Add ToolNet object reference to the instantiated element
until All development objects have been processed

Require: All associations are available in the Relation Repository
repeat
  Get association from the Relation Repository
  Get source ID of the association
  Get IM reference for given source ID
  Get target ID of the association
  Get IM reference for given target ID
  Instantiate IM association between source and target
until All associations have been processed

Figure 4.5: A screenshot of a part of the Navigation Client

evaluation result which resembles the proposed trace Table 3.3. The Visualization Preprocessor passes a list of development object IDs to the Navigation Client. The Navigation Client shows all associations for the objects in a list. The object ID belonging to each association end is also depicted together with a short description. If the development object violates the specified OCL constraint, a red icon appears in the upper left corner of the development object. When the mouse is moved over such a development object, the OCL constraint that has been violated will be shown.

4.4 Summary

The contemporary development process of DaimlerChrysler is to a large extent supported by tools. To support the safety development process, tools exist to assist the developers to create a Safety Argument. However, generation of evidence that has been referenced in a Safety Argument was not supported.

Chapter 3 specified a method which specified what a Safety Argument looks like whose solutions can be generated instead of manually created. While the manual
4.4 Summary A Tool to Generate Solutions

generation of solutions is time consuming and boring, a tool has been developed to automate this process. The tool is named Solution Generator and a prototype of the Solution Generator has been introduced in Chapter 4.

The Solution Generator can automatically generate trace tables for the traceability part of a Safety Argument. The input for a Solution Generator consists of a Safety Argument, development objects, and their associations.

The Safety Argument specifies what the safety data should look like. This information can be extracted from the Safety Argument and converted to Object Constraint Language (OCL) expressions. An OCL engine is able to validate the development data against the OCL expressions.

The Solution Generator can be seamlessly integrated with ToolNet whose services can be reused. A service that has been reused is the Navigation Client. The Navigation Client has been used to represent the trace tables which are generated by the Solution Generator.

A proof of concept of the Solution Generator is given by a real life example system. The tool was able to identify those development objects that violate the constraints specified in the Safety Argument. Finally, trace tables for the system have been generated and presented in the Navigation Client.
Chapter 5

Evaluation

This chapter evaluates both our method and our tool (the Solution Generator). This has been done by means of a case study. Section 5.1 describes first which case has been studied. Section 5.2 and Section 5.3 describe how well the method and tool can be applied to the case.

5.1 EBS Case Study

By means of the Emergency Brake System (EBS) system, our method and tool—presented in Chapters 3 and 4, respectively—will be evaluated. The EBS is a system that can initiate an emergency brake to prevent an accident. Remember that the EBS is an already existing system which has not been developed against the development process steps proposed in Sections 3.2.2.1 and 3.2.2.2. The EBS does not have to be conform to the ISO/WD 26262 safety standard, yet.

A high level subsystem decomposition, showing the systems the EBS is interacting with, has been depicted in Figure 5.1. The EBS subsystem gets data from the distance sensor. When the EBS recognizes a situation that will possibly lead to a collision, it mutes the radio, warns the driver, and when the driver does not react, the system sends a request for an emergency brake to the brake system.

![Figure 5.1: The EBS in its environment](image)

To say show the safety of the structure of the development objects of the EBS, a traceability Safety Argument should be created. For this Safety Argument, evidence can be generated automatically by means of the Solution Generator. To
create a traceability part of a Safety Argument for the EBS, EBS’s development process should be known. As has been described Section 2.2 the following phases have been identified: requirements management, design, implementation, integration, installation, and assessment. Currently, associations only have been created between development objects from the requirements phase, but not between the requirements and objects from subsequent development phases. For example, no associations are available between requirements specified with DOORS and a Matlab/Simulink signal. This means that we did not generate evidence for the whole traceability part of EBS’s Safety Argument.

The identified requirements for the EBS have been decomposed into sub requirements. Table 5.1 lists a number of sub requirements that we analyzed. Furthermore, this table lists the German names for these types and their abbreviation. Notice that the software architecture referenced in the table is a textual requirement to the software architecture of a component.

<table>
<thead>
<tr>
<th>Description</th>
<th>Description in German</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety Requirement</td>
<td>Sicherheitsanforderung</td>
<td>SR</td>
</tr>
<tr>
<td>Safety Concept</td>
<td>Sicherheitskonzept</td>
<td>SC</td>
</tr>
<tr>
<td>System Requirement</td>
<td>Systemanforderung</td>
<td>SYSR</td>
</tr>
<tr>
<td>Component</td>
<td>Komponentenentwicklung</td>
<td>C</td>
</tr>
<tr>
<td>Software Architecture</td>
<td>Software-Architektur</td>
<td>SA</td>
</tr>
</tbody>
</table>

Figure 5.2 provides a high level overview of the associations between the requirements. In this case study, only the safety requirements and safety concepts have been analyzed. A SR can be refined in a safety concept. Also, the definition of a SR can introduce new system requirements. Furthermore, a safety requirement can be directly met by a component or a software architecture object. A SC can also introduce new system requirements. Besides that, it is possible that the SC can be directly met by a component or the software architecture.

Figure 5.3 shows instances of for the requirement types listed in Table 5.1. Please notice that the presented requirements are very generic because DaimlerChrysler does not provide detailed requirements.

To distinguish between the various requirement types, attributes has been added to the requirements. These attributes have been shown in Figure 5.3. In principle, this attribute is superfluous because it is possible to create a separate DOORS module for every requirement type.

The method and the Solution Generator we created, have been applied to the data described in this section. How this has been done, will be described in the next sections.
5.2 Evaluation of the Method

This section shows how a part of the Safety Argument can be constructed. The Safety Argument will be based on the data provided in the previous section.

Step 1: Creating the Safety Argument  The first step of the method which has been applied to the EBS consists of creating the Safety Argument. We decided to verify whether all safety requirements are forward traceable.

The safety requirements of the EBS system can have four possible associations. The first set of associations may consist of links to system requirements, because a safety requirement may sometimes lead up to new system requirements. The second set consists of associations to safety concepts. Safety concepts are refinements of safety requirements which are considered to be too coarse grained. The third part consists of associations to components. Components are EBS’s subsystems, like the radio, or the brake. Finally, there are some direct associations to the software architecture, which can be compared to software requirements. Equation 5.1 expresses that every safety requirement of the EBS should have at least one association to a system requirement, or a safety concept, or a subsystem requirement, or a software requirement or a combination of them. The abbreviations used in the equations have been explained in Table 5.1.

\[
\forall sr SR \rightarrow \exists e (SYSR(e) \lor SC(e) \lor C(e) \lor SA(e)) \quad (5.1)
\]

\[
\forall sc SC \rightarrow \exists e (SYSR(e) \lor C(e) \lor SA(e)) \quad (5.2)
\]

Equation 5.2 shows what a trace should look like for a safety concept. The safety concept has the same associations as the safety requirements, except that a safety concept does not have an association to a safety requirement, while a safety concept a refinement is of a safety requirement.
5.2 Evaluation of the Method

Evaluation

(a) Association between a safety requirement and a safety concept

(b) Example of a system requirement development object

(c) Example of a component development object

(d) Example of a software architecture development object

Figure 5.3: Illustration of data from the requirement specification phase
The decomposition of the realization goal of the Safety Argument has been shown in Figure 5.4. Remember that the pattern that has been used to construct this Safety Argument has been described in Section 3.2.2, of which Figures 3.2 and 3.5 are particularly informative.

![Safety Argument Diagram]

Figure 5.4: A Safety Argument for EBS
Step 2: Creating ToolNet Models  Based on the classes of development objects defined in the Safety Argument, the IM and Tools For Projects Model can be constructed. This is necessary to initialize ToolNet and tell it what is meant by the various requirement types we identified. To create a IM, the development object classes should be taken from the Safety Argument and IM classes should be created for them. Furthermore, associations between the development object classes should be created.

When the development object classes are just added to the model, the IM will become tangled while some relation types have to be defined twice. To overcome this, the auxiliary class GenericSafety has been introduced of which both SafetyRequirement and SafetyConcept inherit. This means that common relations from safety requirements and safety goals to system requirements, components, and software architecture, only have been modeled once. The resulting IM has been shown in Figure 5.5.

Besides the IM, the Tools For Projects Model has to be constructed. The Tools For Projects Model for the safety requirements has been depicted in Figure 5.6. Remember that the safety requirements are formal objects stored in DOORS modules. Each safety requirement can have attributes, for example containing a description. The Tools For Projects Model states that a formal object from DOORS is a safety requirement if the name of the module of that formal object equals “SystemAnforderung” (which is German for system requirement) and the 'Type' attribute—
which describes the type of the formal object—equals “SafetyRequirement”. See Figure 5.3a for an example.

Figure 5.6: The Tools For Project Model for a SafetyRequirement

When new systems will be defined, we recommend that every development object class in DOORS gets its own module. For example, safety requirements and safety concepts should not be a part of the module named SystemRequirement. This makes it easier to understand the project structure and it is not necessary anymore to give each formal object a 'Type' attribute. Notice that the logical equations from the Safety Argument do not reference the GenericSafety class because it is just an auxiliary class which can be removed without losing information.

**Step 3: Creating Solutions** Manually creating the solutions from the data, the associations and the Safety Argument is time consuming. All associations for the EBS safety requirements and safety concepts should be examined and the results should be written to a list. Chapter 4 describes a method that can generate the solutions automatically. With the help of the tool, the solutions will be generated for the running example.

**5.3 Evaluation of the Solution Generator**

Applying the evaluation method to the EBS, described in Section 5.2, results in a Safety Argument and ToolNet models. Manually evaluating the solutions turned
out to be too time consuming. However, once the Solution Generator has been implemented, solutions for the given system can be generated automatically.

This section shows how the Safety Argument for the EBS can be evaluated with the help of the Solution Generator. Therefore, the logical expressions need to be converted to OCL constraints. Furthermore, the tool needs the development objects and their associations. While the EBS is a system that already exists, this data has already been created. Once the Safety Argument has been constructed, it becomes possible to create OCL constraints and generate the trace tables for the EBS. All steps will be described in the subsequent sections.

### 5.3.1 Creating OCL Expressions

To convert the logical equations (Equations 5.1 and 5.2) to OCL expressions, it is not possible to use lookup Table 4.3 while a different pattern has been described by this table. That means that the lookup table has to be extended. Equation 5.1 expresses that a safety requirement should have an association to a system requirement, a safety concept, a component, or a software architecture element. Equation 5.2 can be analyzed in the same way. Listing 5.1 shows an OCL file, where lines 3 till 7 represent the OCL constraint for Equation 5.1 and the remainder of the file presents the constraint for Equation 5.2.

<table>
<thead>
<tr>
<th>Relation between</th>
<th>Name of the association</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety Requirement and Safety Concept</td>
<td>safetyConcept</td>
</tr>
<tr>
<td>Safety Requirement and System Requirement</td>
<td>systemRequirement</td>
</tr>
<tr>
<td>Safety Requirement and Components</td>
<td>component</td>
</tr>
<tr>
<td>Safety Requirement and Software Architecture</td>
<td>softwareArchitecture</td>
</tr>
</tbody>
</table>

Table 5.3: Lookup table for patterns of logical expression specified in EBS’s Safety Argument

<table>
<thead>
<tr>
<th>Safety Argument Pattern</th>
<th>OCL expression</th>
</tr>
</thead>
</table>
| \( \forall a . A \rightarrow \exists b . (X(b) \lor Y(b))^* \) | context A
inv: \( \text{self.x->size()} >= 1 \)
[or \( \text{self.y->size()} >= 1 \)]^* |

Listing 5.1: OCL expressions for EBS

```java
package Projects::SafetyCase::Requirements::
TextbasedRequirementsEngineering
```
5.3 Evaluation of the Solution Generator

```
context SafetyRequirement
inv: self.safetyConcept->size() >= 1 or
    self.softwareArchitecture->size() >= 1 or
    self.component->size() >= 1 or
    self.systemRequirement->size() >= 1

context SafetyConcept
inv: self.softwareArchitecture->size() >= 1 or
    self.component->size() >= 1 or
    self.systemRequirement->size() >= 1
endpackage
```

Line 1 and 14 show some OCL engine specific code which has been explained in [8]. In the context field of both OCL expressions (line 3 and 9) the antecedent of the logical equations is represented, being SafetyRequirement and SafetyConcept. From these classes, the associations to other classes have been defined. The association classes are all listed in the consequent of the logical equations. The relations to other classes all have a name which has been specified in the IM. See Figure 5.5 for EBS’s IM.

5.3.2 Analysis’s Results

Figure 5.7 shows the result of a run of the Solution Generator applied to the EBS. The results are presented in ToolNet’s Navigation Client.

On the left side of the Navigation Client the safety requirements and safety concepts have been listed. The IDs and a short description of the development objects has been listed. The associated development objects for every safety requirement and safety concept have been depicted. When a development object has been marked by an icon with a cross, that object violates an OCL constraint. This indicates that some associations are missing. By scrolling over the development object that violates a constraint, that OCL constraint is shown.

The Navigation Client shows a popup that summarizes the results of the trace table generation. The report shows that 61 development objects have been analyzed. These development objects are the ones specified in Table 5.1. From these 61 objects, 25% violates the specified OCL constraints. Based on this information, managers can estimate how long it will take to make the system acceptable safe.

To help developers to navigate to the development objects listed in the Navigation Client, it is possible to click on a development object and jump to that particular development object in a ToolNet Enabled Tool. In this way, the missing associations can be drawn quickly.
5.3 Evaluation of the Solution Generator

Figure 5.7: A trace table shown in ToolNet’s Navigation Client

5.3.3 Consistency

It is important to realize that the IM MDR provides a static view on the development data and their associations. This implies that when a development object in a tool, or an association in ToolNet changes, this change will not be reflected in the IM MDR. Only when the IM MDR will be reconstructed, the modified data will be added to the IM MDR.

The static view on the data is desired because performing an analysis of the real time data can cause inconsistencies. Consider an OCL engine that has real time access to the data and assume that development objects $a$ and $b$ should be and have been associated to one another. When the OCL engine inspects these development objects, the association will be found and the OCL expression will evaluate to true. This result will be displayed in the Navigation Client. However, when a developer deleted the association just before the result is displayed in the Navigation Client, the Navigation Client shows an incorrect result. To prevent this from happening, a static view on the data has been defined which is not meant to be edited.
5.3.4 Scaleability

The presented case study shows the generation of trace tables for two OCL constraints. However, a complete argument can consist of more than one hundred constraints. This section shows that the Solution Generator is able to handle this much data.

**Time Complexity** The generation of the trace tables for the case study took 270 seconds. Figure 5.8 shows how long it took for the various components of the Solution Generator to execute their tasks. As can be seen, the bottleneck is the creation of the IM MDR which took about 93% of the time.

![Figure 5.8: Comparison of the execution time of the Solution Generator’s components](image)

A further analysis of the IM MDR reveals that getting the data from the tools takes the most time. As can be seen in Figure 5.9, it takes approximately two minutes to get the date export of the SoftWareArchitecture. Adding the data from the export to the IM MDR took about five seconds. Because the export of the data
is requested via standard ToolNet functionality, this procedure cannot be optimized by changing the Solution Generator.

![Diagram showing time taken for different specifications](image)

**Figure 5.9:** Time it takes to get the exports from the data sources and instantiate them in the IM MDR

Figure 5.9 also shows that there are vast differences in the time it took to get data exports. This is largely dependent on the number of development objects comprised by a certain data source. For example, the SoftWareArchitecture export
contains approximately 26 times more development objects than the instrument’s export (INS). Besides that, if a development object contains an image it will take much more time to get the data.

Because creating the IM MDR takes a lot of time, the MDR has been made persistent. This means that if the data does not change, the database can be used over and over again. This will speed up the generation of the trace tables considerably. When the IM MDR is reused, it will only take twenty seconds to calculate and display the trace tables which increases the performance by 14 times.

To handle very large data sets, it is recommended to create and store an IM MDR. The database can be reused to perform various kinds of analyses on the data. This will speed up the analysis process, because—as has been made clear—creating the IM MDR consumes the most time. Doing so, it will be possible to create the IM MDR over night and perform analyses on it later on.

When a large set of OCL constraints is specified, it might be wise to distinguish several subsets. These constraint subsets can be validated one by one, which will also reduce the execution time. For instance, by first generating the trace tables for the development objects created during the requirements phase, it will be possible for developers to create the missing associations. This process can be repeated for every development process step.

**Space Complexity** The IM MDR that has been created for the case study has a size of 770 KB. While only important information about the development objects is stored, for example the ID and the description, the database will not grow too large. By only storing important information, it is possible to store large amounts of development objects in the MDR.

**5.3.5 Modifiability**

The Solution Generator relies heavily on ToolNet. However, when an enterprise does not use ToolNet—but uses another data integration approach, instead—can the concept of the Solution Generator still be used? We think that this is possible if the other integration approach supports the definition of development objects. Without the notion of a development object, it would be impossible to reference development data in the Safety Argument. Furthermore, the approach should somehow be able to show an integrated view on the data and their associations. When these requirements are fulfilled, it should be possible to analyze the data based on constraints specified in the Safety Argument. When the database that contains the integrated view on the development data provides a JMI interface (see Section C.4), it would even be possible to use the OCL engine which has been used in the Solution Generator.

**5.3.6 Correctness and Completeness**

The objective of this thesis was to show whether it is possible to automatically generate evidence for parts of the Safety Argument. The Solution Generator has
proved that this is possible. However, we did not pay special attention to prove correctness and completeness of the tool. The reason is that quite a lot of legacy code from ToolNet and the OCL engine has been used. Proving that this legacy code complies with a SIL is not easy.

As long as the ISO/WD 26262 has not been ratified, it is not necessary for manufacturers in the automotive industry to create a Safety Case. In this situation, the Solution Generator can be used to trace flaws in the system and it is not critical to have a tool that is SIL compliant. However, as soon as regulation authorities decide that a Safety Case has to be created for systems in the automotive industry, it will be necessary to prove the correctness and completeness of the tool. See also Section 6.3 about this topic.

5.4 Summary

Both the method and the tool have met their objectives. The method specifies how a traceability part of the Safety Argument can be constructed. Furthermore, trace tables have been described that give evidence for a particular solution. These trace tables will be constructed by means of development data and associations which are accessible via ToolNet. Also, a description has been given how the IM can be constructed from the Safety Argument.

Once the IM has been constructed, the Solution Generator can generate trace tables. It turned out that the tool has been implemented according to the requirements which has been defined in Chapter 4. Although the Solution Generator relies on ToolNet, it is possible to use other tool integration techniques as basis to generate trace tables. To prevent that the Solution Generator becomes inconsistent, a static view on the development objects and their association has been defined. In contrast to a dynamic view, the static view is not meant to be changed by developers.

Both time and space complexity of the tool seem to be manageable, even if a large amount of analyses are performed on a large amount of data. By storing the IM MDR, the performance can be increased significantly.

Not much effort has been put in proving correctness and completeness of the Solution Generator, while quite a lot of legacy code has been reused for which it is not straightforward to prove correctness and completeness. The correctness and completeness of the Solution Generator should be proved before the Solution Generator can be used to convince TÜV inspectors that the traceability part of the Safety Argument is safe.

All in all, according to the metrics defined in this chapter, both the method and the Solution Generator are functioning properly.
Chapter 6

Conclusion

A summary of this thesis will be given in Section 6.1. Furthermore, a conclusion for this project will be described in Section 6.2. This conclusion shows what we found out during this thesis project. A reflexion on these results will be given in this section, too. Finally, Section 6.3 lists items that have not been studied yet. These topics should be studied in subsequent research projects.

6.1 Summary

The objective of this thesis was to find out how a Safety Argument looks like whose evidence can be generated automatically. Furthermore, a tool has been developed that can generate this evidence automatically.

We found out that traceability can be proven automatically. Traceability means that the origin, realization and proof for a requirement are clearly described in the documentation. The origin and realization of the safety requirements have been studied during this thesis.

Showing the origin of the safety requirements means that every safety requirement should be connected to a hazard via a safety goal. This has been referred to as backward traceability. In the same way, all hazards should be connected to safety requirements. This is called forward traceability. Documenting the origin of safety requirements ensures that all hazards have been considered in the safety requirements. Furthermore, there will not be safety requirements that have not been connected to a hazard which ensures that no unnecessary, incorrect functionality is introduced in the system.

The realization of the safety requirements ensures that all safety requirements have been implemented in the final system. This means that the safety requirements should be traced through all development stages. A typical development process consists of the following stages: requirements engineering, design, implementation, integration, and assessment. Again, forward and backward traceability should be checked.
6.2 Conclusions

Trace tables are used to show that safety requirements have been implemented in a particular stage. When this procedure is repeated for all development stages, a prove has been given that the inspected system is traceable.

The next thing is to find a method that specifies how trace tables can be generated. First, the Safety Argument should have been created. Second, the development data and the associations between them should be accessible.

The Safety Argument should be enriched with a branch containing traceability information. This means that expressions in the Safety Argument state which traceability conditions must hold for the described system, e.g., all hazards should be associated to a safety goal. However, mentioning hazards and safety goals in the Safety Argument does not mean that it is clear what is meant by these terms. To overcome this problem, a mapping should be made between the entities referenced by the Safety Argument—also termed development objects—and the actual development data. This mapping can be made with the help of the traceability tool ToolNet.

ToolNet defines development objects with the help of an Information Model (IM). This IM contains all development objects and associations referenced in the Safety Argument. In other words, the IM models the traceability relations between development objects of a particular system. When the IM is finished, the development objects of the system are uniquely defined.

Once the Safety Argument and the IM have been constructed, the evidence should be generated. This comes down to validate the development data against the traceability information of the Safety Argument. The results of this validation should be written to trace tables. Manually creating these tables is time consuming because contemporary systems comprise a lot of data. Therefore, a tool named Solution Generator has been created to generate the tables.

The Solution Generator’s SAtoOCL component reads the traceability expressions from the Safety Argument. These statements are converted to the Object Constraint Language (OCL). This language makes it possible to define constraints for the IM. An OCL engine is able to validate the OCL expressions against the development data. A database with an integrated view on all development objects and associations is created by the Solution Generator’s IM MDR component.

Finally, the results generated by the OCL engine needs to be presented in trace tables. The Visualization Preprocessor component prepares the raw results from the OCL engine for presentation as trace tables. The trace tables are presented in ToolNet’s Navigation Client.

6.2 Conclusions

In this chapter we show that we were able to formulate answers to the research questions stated in Chapter 1.
Conclusion

What is meant by a Safety Case, considering the context of the automotive industry? We were able to find a definition of the term Safety Case because we thought the definition from the ISO/WD 26262 was not precise enough. Besides a definition, a way to define the argument—which forms an essential part of the Safety Case—has been found, too.

What does a method look like that is able to generate a Safety Case? The method describes how the traceability part of the Safety Argument can be constructed. Although some reusable Safety Argument structures have been specified by Kelly, we have not found a structure that shows how to deal with traceability. We think that an evaluation of the traceability will contribute to the safety of systems because it can be ensured that all hazards and safety requirement can be tracked through all development stages, ensuring that a particular requirement is apparent in the final system.

What does a tool look like that automates the generation of Safety Cases? According to us, automatically evaluating traceability has a huge advantage; developers are not willing to do this manually, because it is boring and takes a lot of time. In our opinion, the attempt to automatically generate evidence for Safety Cases is unique. In the future, it might be possible to generate more evidence automatically which will reduce the production cost and increase the safety of the systems that have to be developed.

Which metrics can be defined against which the method and tool can be validated and what is the result of such an evaluation? As we have pointed out in the previous chapter, the method and tool have met their objectives. However, we think that it will be very important to show that the Solution Generator conforms to a SIL.

6.3 Future Work

This thesis does not address all issues related to the automatic generation of evidence for Safety Cases. Further research is needed to solve these problems.

1. Safety of the Solution Generator: When a system will be developed according to a particular SIL, then the tools used for the generation of evidence should also be developed against this level. This ensures that such a tool functions correctly. The ISO/WD 26262 part 8: "Supporting Processes" Chapter 13: “Qualification of software tools” defines requirements for such tools.

Currently, both the Solution Generator and ToolNet have not been developed against a SIL. To make the Solution Generator commercially available, it should be proved that all development objects and associations are available in the IM MDR. Besides that, the OCL engine should be correct and the
6.3 Future Work

SAtoOCL component should convert the equations from the Safety Argument to the right OCL constraints.

2. **Automatic generation of the IM and SAtoOCL component:** At the moment of writing, the creation of the IM has to be done manually. All development objects referenced in the Safety Argument should be transferred to the IM. For a Safety Argument with many solutions this conversion can be difficult. It is possible to generate the IM automatically with the help of, for example, Triple Graph Grammars [28] but other model to model engines probably can be used, too.

The SAtoOCL component has not been implemented, yet. Instead, the conversion from expressions in the Safety Argument to OCL constraints has to be done manually. Before implementing the SAtoOCL component, it might be interesting to read Section C.5.

3. **Maintenance of the Safety Case:** The main focus of this thesis has been on the creation of an initial Safety Argument which is created before, or during the development process. However, most of the time, some flaws in the system become apparent when the system has been finished and sold. Of course, these flaws should be repaired. Such an improvement has impact on the Safety Case but we did not study what the results are of such an improvement.

4. **Safety Analyzer:** To make the Solution Generator a commercial success, we think that it will be necessary to provide more information for managers. Currently, only information has been given about the number of leaf goals that are true or false. However, more information could be provided to managers. For instance, the expected time it takes to make the system acceptably safe. This prediction can probably be given based on data gathered in history. The relation between the generated evidence and the Safety Analyzer is given in Appendix E.

5. **Evaluate a whole system:** The case study of this thesis concentrates mainly on the requirements phase of the development process. This is because associations between other phases of the development process are subject to further research and have not been created for the example EBS.

6. **Which parts of the Safety Argument can be generated:** This thesis describes how solutions can be generated for a part of the traceability Safety Argument. However, it might be possible to generate more solutions automatically, an example has been shown in Appendix B.

7. **OCL engine:** The OCL engine which has been used for the Solution Generator, is a prototype from the TU Berlin. It should be validated that this engine is complete and correct. When not, the engine should be repaired or replaced.
According to us, some of the topics listed above should be investigated before the product can be brought to market. First of all, the correctness of the Solution Generator (and ToolNet) should be proved. Although the product can be used in the automotive industry as long as the ISO/WD 26262 has not been ratified. The tool provides useful traceability information to developers. By considering this information, systems can be made safer, although the Solution Generator has not been developed against a SIL, yet.

Besides a proof of correctness of the Solution Generator, it should be proved that requirements can be traced through the whole system. Although, the concept of the Solution Generator can theoretically be scaled to the whole development process, this should be verified in practice. Finally, the OCL engine should be analyzed. If the OCL engine is not correct, it should be replaced or repaired.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Glossary

A

Acceptable Safe  A selected threshold level below which safety cannot be guaranteed.

Accident  An undesired and unplanned (but not necessarily unexpected) event that results in (at least) a specified level of loss.

Association  A traceability link between two development objects.

D

Development Object  A (MOF M1) instance of a class defined in the IM (at MOF level M2).

Development Tools  Any tool used during the development process of embedded safety related systems at DaimlerChrysler.

E

ECU  See Electronic Control Unit.

Electronic Control Unit  A part of a system consisting of hardware and software.

Embedded System  A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function.

Emergency Braking System  A system that is able to automatically make an emergency brake if a collision is about to occur.

Extensible Markup Language  A language that is often used for specifying documents.
Goal Structuring Notation  A notation to define safety cases.

G

GSN   See Goal Structuring Notation.

H

Hazard   A state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event).

I


IM   See Information Model.

IM MDR   An Information Model Meta Data Repository, i.e., a MDR that holds an IM model and instantiated classes and associations of this model. The instances represent the development data from the various development tools.

Information Model   A model that defines which objects can be referenced for a particular project. This model is used by ToolNet to describe data at MOF level M2.

ISO/WD 26262   A working draft of a safety standard titled “Road vehicles Functional safety”.

J

Java Metadata Interface   An interface that describes a generic way to access data from the MDR.

JMI   See Java Metadata Interface.

L

Leaf Goal   A GSN goal that only has solution as children.

Logical Equation   Logical equations used in the Safety Argument to specify associations.
GLOSSARY

M

MBD  See Model-Based Development.

MDR  See Metadata Repository.

Meta Object Facility  The MOF defines an architecture for describing and defining
data and meta data.

Metadata Repository  An object oriented database that is able to store MOF meta
models and its data.

Model-Based Development  In Model-Based Development, the model is the cen-
tral artifact and is used and systematically refined throughout the entire
development process which is literally based on or centered around it.

MOF  See Meta Object Facility.

N

Navigation Client  A ToolNET component that has been used to visualize trace
tables.

P

Pattern  A pattern is a generic logical expression which instances are used in the
leaf goals of the Safety Argument.

R

Relation Repository  A ToolNET component that stores the traceability links be-
tween various development objects.

Requirements Satisfaction  Demonstration that all Safety Requirements have been
met.

Requirements Traceability  Demonstration that all Safety Requirements have been
tracked throughout all stages of System Development and Safety Anal-
ysis.

Requirements Validation  Demonstration that the set of Safety Requirements is
complete and accurate.

S

Safe  The freedom from accidents or losses.

Safety Argument  Definition of a Safety Case argument specified in the GSN.
**Safety Case**  A Safety Case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context.

**Safety Integrity Level**  Four levels that specify the number of failures per hour a system may initiate.

**SIL**  See Safety Integrity Level.

**Solution**  An element from the GSN which references evidence.

**Tool Model**  A ToolNet model that models the information stored in the development tools.

**ToolNet**  A tool that is used for the creation of traceability links between development objects.

**ToolProxy**  ToolNet component that stores data in off line mode.

**Tools For Projects Model**  A ToolNet model that defines a mapping between the Tool Model and the IM.

**XML Metadata Interchange**  A description on how to convert MOF compliant models into XML.

**XML**  See Extensible Markup Language.
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>accident, 1</td>
<td></td>
</tr>
<tr>
<td>AntiPattern, 13</td>
<td></td>
</tr>
<tr>
<td>CTE, 19</td>
<td></td>
</tr>
<tr>
<td>Development Object, 19</td>
<td></td>
</tr>
<tr>
<td>DOORS, 18</td>
<td></td>
</tr>
<tr>
<td>EBS, 3</td>
<td></td>
</tr>
<tr>
<td>ECU, 15</td>
<td></td>
</tr>
<tr>
<td>Electronic Control Unit, 15</td>
<td></td>
</tr>
<tr>
<td>Electronic Safety Case, 10</td>
<td></td>
</tr>
<tr>
<td>Embedded System, 14</td>
<td></td>
</tr>
<tr>
<td>Emergency Brake System, 3</td>
<td></td>
</tr>
<tr>
<td>Goal Structuring Notation, 11</td>
<td></td>
</tr>
<tr>
<td>GSN, 11</td>
<td></td>
</tr>
<tr>
<td>Hardware and Software Design phase, 17</td>
<td></td>
</tr>
<tr>
<td>Hardware Requirement, 16</td>
<td></td>
</tr>
<tr>
<td>Hazard, 16</td>
<td></td>
</tr>
<tr>
<td>IEC 61508, 2</td>
<td></td>
</tr>
<tr>
<td>IM, 19</td>
<td></td>
</tr>
<tr>
<td>IM MDR, 48</td>
<td></td>
</tr>
<tr>
<td>Implementation phase, 17</td>
<td></td>
</tr>
<tr>
<td>Integration and Assessment phase, 17</td>
<td></td>
</tr>
<tr>
<td>Integration Component, 44</td>
<td></td>
</tr>
<tr>
<td>ISO/WD 26262, 2</td>
<td></td>
</tr>
<tr>
<td>MBD, 16</td>
<td></td>
</tr>
<tr>
<td>MDR, 45, 90</td>
<td></td>
</tr>
<tr>
<td>Meta Data Repository, 45, 90</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Based Development, 16</td>
<td></td>
</tr>
<tr>
<td>Navigation Client, 45</td>
<td></td>
</tr>
<tr>
<td>Object Constraint Language, 47</td>
<td></td>
</tr>
<tr>
<td>OCL, 47</td>
<td></td>
</tr>
<tr>
<td>OCL engine, 48</td>
<td></td>
</tr>
<tr>
<td>Polyspace, 19</td>
<td></td>
</tr>
<tr>
<td>Product Model, 27</td>
<td></td>
</tr>
<tr>
<td>Project Information Model, 19</td>
<td></td>
</tr>
<tr>
<td>Prolog, 46</td>
<td></td>
</tr>
<tr>
<td>QAC, 19</td>
<td></td>
</tr>
<tr>
<td>Quality Center, 19</td>
<td></td>
</tr>
<tr>
<td>Relation Repository, 45</td>
<td></td>
</tr>
<tr>
<td>Requirements Specification phase, 15</td>
<td></td>
</tr>
<tr>
<td>safety, 1</td>
<td></td>
</tr>
<tr>
<td>Safety Argument, 11, 27</td>
<td></td>
</tr>
<tr>
<td>Safety Case, 2, 9</td>
<td></td>
</tr>
<tr>
<td>Safety Case Pattern, 13</td>
<td></td>
</tr>
<tr>
<td>Safety Case Report, 10</td>
<td></td>
</tr>
<tr>
<td>safety critical system, 1</td>
<td></td>
</tr>
<tr>
<td>Safety Goal, 16</td>
<td></td>
</tr>
<tr>
<td>Safety Integrity Level, 10</td>
<td></td>
</tr>
<tr>
<td>Safety Requirement, 16</td>
<td></td>
</tr>
<tr>
<td>SAtoOCL, 48</td>
<td></td>
</tr>
<tr>
<td>SIL, 10</td>
<td></td>
</tr>
<tr>
<td>Simulink, 18</td>
<td></td>
</tr>
<tr>
<td>Software Requirement, 16</td>
<td></td>
</tr>
</tbody>
</table>
Solution, 34
Solution Generator, 41
SQL, 46
Stateflow, 18
Structured Query Language, 46
Subsystem Requirement, 15
System Design phase, 16
System Requirement, 15

TargetLink, 18
TGG, 38
Therac-25, 1
Tool Model, 20
ToolNet Enabled Tool, 44
ToolProxy, 45
Tools For Projects Model, 20
Top Level Requirement, 15
TPT, 19
Traceability, 4
Triple Graph Grammar, 38
Type Hint, 21

unacceptable risk, 1

Visualization Component, 44
Visualization Preprocessor, 48
Appendix A

The Goal Structuring Notation

Figure A.1 shows which symbols have been defined for the Goal Structuring Notation (GSN).

Figure A.1: Overview of symbols used in the GSN

Goals, sub goals and leaf goals are used to represent claims. Leaf goals are connected to solutions which reference evidence. A strategy node can be used to
The Goal Structuring Notation

explain why a goal or sub goal has been decomposed. Together, the goal, strategy, and solution nodes represent the spine of the Safety Argument. The other nodes are used to document the rationale of the spine nodes.
Appendix B

Extended Evaluation Method

This appendix provides additional information about the decomposition of the “Proof of Safety requirements” shown in the Safety Argument depicted in Figure 3.2. Furthermore, a short overview will be given about what we found out about the Traceability Report and the Colored Safety Argument as output of the method defined in Figure 3.1. While an overview of these topics is given in Chapter 3, it is strongly recommended to read this chapter before reading this Appendix.

At this moment, proving the safety requirements cannot be done automatically because that problem is too complex. Therefore, humans are needed that verify the safety requirements. This is done by means of reviews and test. Reviews are documents that contain comment of a developer to a certain development object. Some development objects are even evaluated by multiple developers. Test artifacts consist of test specifications, an implementation and results of the tests. Notice that these tests artifacts are also subject to reviews.

Constructing a traceability Safety Argument for the proof goal will be shown in Section B.1. Once the Safety Argument has been finished, solutions can be generated in the same way as explained in Chapter 4. Furthermore, the Safety Argument together with the solutions can be written to a traceability safety report (Section B.2).

B.1 Extended Safety Argument

The prove goal of the Safety Argument pattern depicted in Figure 3.2 has two sub goals, being “The safety requirements have been validated” and “The safety requirements have been satisfied”. Weaver [49] states that validation means that the should be demonstrated that the set of safety requirements is complete and accurate. Satisfaction is defined as the demonstration that all safety requirements have been met.

According to us, validating the safety requirements can be best done with the help of existing hazard analysis methods, like FTA or HAZOP. This has not been described further. However, demonstrating that all safety requirements have been
met can be done by reviews and tests. Therefore, new subgoals can be added to the satisfaction subgoal. These subgoals can be filled with Equations B.1 to B.3.

As stated above, humans are needed to check the validity of development objects. The results will be written in traceability reports which are usually stored in DOORS. As stated by Dörr [12], a review report may, among others, consist of checklists, or comments from domain experts. A review document typically contains the comment of the expert, a status of the review, and a criticality of the fault that has been found.

Consider the Review sets shown in Figure B.1. Here, the review documents have been depicted as red squares. However, unlike the member of the sets shown in Figure 3.3, the members of Review have a vector which shows an attribute for that particular member. For brevity, the size of the attribute vector is one and only the status of the review is shown. From the figure, one can conclude that two results have been set to ‘ok’, meaning that the development reviewed in the report is considered to be valid. However, one result is ‘open’, indicating that the report contains a comment that states that the associated development object is not valid. The same holds for the Tests. The elements of $B$ have been tested for example, because members of $B$ represent implementation elements for which a SiL test has been performed. The result of such a test should also be ‘ok’.

Besides the fact that the members of the Review and Test sets have associated attribute vectors, Figure B.1 makes clear that every review and test have been connected to a development object. The development objects can be represented in the same way as in Figure 3.3. The associations between the review and test documents and the development objects from $A$ and $B$ have been represented as a dotted red lines.

To construct the Safety Argument for reviews and test, it is not sufficient to check for traceability. Stating that every development object has a review—except the reviews itself, although there is a quality review which checks the quality of the reviews, which has not been studied in detail—does not mean that every develop-
Extended Evaluation Method

B.1 Extended Safety Argument

ment object is valid, since the status of the review may be 'open'. Therefore, not only the association, but also the results of the review should be verified. How this can be done is expressed by Equation B.1 and B.2, where RA and RB denote the ReviewA and ReviewB, respectively. Notice that it is possible to add an equation for every set of development objects, for example hazards, safety goals, etcetera. The element from B should also have a link to a test element which attribute should also be true. This has been shown in Equation B.3, where T denotes the set of tests.

\[
\forall aA \rightarrow \exists revRA \land \forall revRA(rev.result == ok) \quad (B.1)
\]

\[
\forall bB \rightarrow \exists revRB \land \forall revRB(rev.result == ok) \quad (B.2)
\]

\[
\forall bB \rightarrow \exists tT \land \forall tT(t.result == ok) \quad (B.3)
\]

Notice that for reviews and tests, nothing is said about forward or backward traceability while this is irrelevant. Reviews and tests that are not associated to a development object do not cause a hazard because they will not be in the final system. However, the claim is that all review and test results in a review set should be ok.

B.1.1 The Evaluation Method

Once the logical expressions have been defined against which the project data can be checked, these equations can be transfered to the Safety Argument.

Solutions Again, the solutions can be given in the form of tables. Every leaf sub goal shall have two solutions, a trace table and an attribute table. A trace table (for the part of the equations resembling the pattern \( \forall qQ \rightarrow \exists sS \)) can be created with shows lists of all elements of the antecedent. Then, at least one review element should be listed for that equation. The attribute table (for the part of the equations resembling the pattern \( \forall sS(s.result == ok) \)), a list of all reviews and tests should be given with the value of the result attribute.

The trace table will look the same as the those described in Section 3.2.3. An example attribute table has been shown in Table B.1.

<table>
<thead>
<tr>
<th>rev ID</th>
<th>Description</th>
<th>Evaluation</th>
<th>Value of the result attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev_1</td>
<td>“description...”</td>
<td>false</td>
<td>open</td>
</tr>
<tr>
<td>rev_2</td>
<td>“description...”</td>
<td>true</td>
<td>ok</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>rev_{n-1}</td>
<td>“description...”</td>
<td>false</td>
<td>open</td>
</tr>
<tr>
<td>rev_n</td>
<td>“description...”</td>
<td>false</td>
<td>open</td>
</tr>
</tbody>
</table>
B.2 Traceability Report

Constructing the IM  Section 3.3.1 described a way to create an IM for a trace table sub goal. This approach can be reused to construct the IM for the newly introduced trace table solution. However, somehow the information from the attribute table solution should also be added to the IM.

Basically, every attribute referenced in the solutions should be added to the development object class it belongs to. This is no problem, because the IM has been drawn in UML, which can represent class attributes.

Notice that only creating the IM is not enough. The development objects in the tools should also represent the IM attributes, otherwise the IM is not modeling reality. For the example reviews, this means that a DOORS attribute representing the result of a review should exist.

Automatic Creation of the Solutions  Once the IM has been constructed and has been enriched with attribute information, it becomes possible to create the solutions in an automatic way. At the moment of writing, ToolNet is not able to deal with attribute information in the IM. Therefore, constructing a IM with attributes is not useful at the moment.

When it is possible for ToolNet to deal with an attributed IM, the attributes from the development object elements will become available. Then, it is possible to ask ToolNet for a particular attribute value of a development object. This information, in turn, can be written to the attribute table. The information in the table can be used by developers to locate reviews, tests, and other development objects that need to be changed.

B.2 Traceability Report

A Safety Case report should, among others, provide an argument why the described system is acceptably safe [5]. As shown in Chapter 3, traceability is a part of the Safety Argument and the Traceability Report is that part of the Safety Case report that proves that the system is safe with respect to traceability. The report can be used to convince safety inspectors that the system is acceptably safe.

A typical traceability Safety Argument contains many goals and solutions and representing the argument in a clear way is a challenge, especially when the report should be a paper document. In such a document, the pages are usually relatively small, and it will be necessary to compress the GSN structure till it fits on a page, or, the argument should be split into several subparts. Both options will normally make it more difficult to easily comprehend the argument.

A completely different approach can also be chosen. This solution does not print the traceability argument on paper, but it rather keeps it electronic, i.e., in computer memory [9]. The advantage of an Electronic Safety Case over a Safety Case written on paper is evident. The argument will not be scattered over several pages, but is, as a whole, stored in a dedicated Safety Case tool. Furthermore, it is easier to
navigate through the large amount of material. For instance, hyperlinks can be used to reference key documents that contain evidence.

Our proposed method and tool can be integrated seamlessly with the electronic Safety Case, in particular the part of the traceability report. There already exist tools to construct a Safety Argument in GSN. These tools serve as basis for the electronic document, however, the tools should be extended. One extension is to make a button which, when it will be pressed, generates the solutions (for which the tool described in Chapter 4 can be used) and then generates the Colored Safety Argument (as described in Section E.2). Based on the Colored Safety Argument, some statistical data can be calculated for managers, for example an estimation can be given about the time it will take to complete the traceability part of the system. Furthermore, the tool can be extended to support hyperlinks, making it possible to create links to generated trace and argument tables. This is helpful for developers who have to create associations between development objects which did not have associations according to the evaluation. The tables provide an excellent overview of development objects that violate the associated claim. When a developer wants to navigate to a development object, he can click on the row of the table, containing the development object. Then, the ID of the development object can be passed to ToolNet, and with existing functionality, the relevant development tool can be opened and the development object element can be selected.

An electronic report will make it easier for managers and developers to handle the large amount of development data. Furthermore, development object elements without associations can be identified more quickly. When the Safety Case for the automotive industry will be realized in this way, it certainly can make the development process more cost effective!
Appendix C

Used Concepts

This appendix provides more information on concepts that have been used in Chapter 4. First, Prolog and SQL examples have been given in Section C.2 and Section C.3 respectively. These examples show what is needed to evaluate the Safety Argument with the respective technique. This makes it easier to understand why these techniques have not been used in the Solution Generator. An example data set that will be used in the Prolog and SQL example will be presented in Section C.1.

Second, the MDR will be introduced. The IM MDR is a database that holds both development objects and their associations. The OCL engine uses the IM MDR for its analysis. Section C.4 will explain the functions of the MDR.

Third, information is provided about a format to which the Safety Argument can be exported. This export format should simplify the task of the SAtoOCL component of the Solution Generator. Ins and outs of the export format will be described in Section C.5.

C.1 Example Data Set

The following example data set will be used in Section C.2 and Section C.3. There is a set of safety requirements, software requirements, and tests. These development objects have been listed in Table C.1. The table only lists the IDs of the development objects and not their description. However, getting these descriptions is not a problem because ToolNet has been used to implement the Solution Generator. ToolNet is able to get the descriptions for a given development object ID.

Table C.2 shows which development objects have been associated to one another. Table C.2a shows that safety requirement $srA$ has been connected to two software requirements $swrA$ and $swrB$. Furthermore, $srB$ has been connected to $swrC$. In the same way, Table C.2b defines relations between software requirements and tests.

Besides creating development objects and associations between them, it is necessary to define a Safety Argument for the listed data. The logical expressions from the Safety Argument are shown in Equation C.1 and C.2, where $SR$ denotes the set
C.2 Prolog

Used Concepts

<table>
<thead>
<tr>
<th>SafetyRequirement</th>
<th>SoftwareRequirement</th>
<th>Test</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>srA</td>
<td>swrA</td>
<td>testA</td>
<td>ok</td>
</tr>
<tr>
<td>srB</td>
<td>swrB</td>
<td>testB</td>
<td>open</td>
</tr>
<tr>
<td>srC</td>
<td>swrC</td>
<td>testC</td>
<td>ok</td>
</tr>
<tr>
<td>srD</td>
<td></td>
<td>testD</td>
<td>open</td>
</tr>
</tbody>
</table>

(a) Safety requirements

(b) Software requirements

(c) Tests

Table C.1: Example set of development objects

<table>
<thead>
<tr>
<th>SafetyRequirement → SoftwareRequirement</th>
<th>SoftwareRequirement → Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>srA → swrA</td>
<td>swrB → testA</td>
</tr>
<tr>
<td>srA → swrB</td>
<td>swrC → testB</td>
</tr>
<tr>
<td>srB → swrC</td>
<td></td>
</tr>
</tbody>
</table>

(a) Associations between safety and software requirements

(b) Associations between software requirements and tests

Table C.2: Sets of associations between development objects

of safety requirements, $SWR$ the set of software requirements, and $T$ the set of tests. See Appendix B for more information about the pattern shown in Equation C.2.

\[
\forall srSR \rightarrow \exists swrSWR \quad (C.1)
\]

\[
\forall swrSWR \rightarrow \exists tT \land \forall tT(t.result == ok) \quad (C.2)
\]

Manually evaluating Equation C.1 shows that $srC$ and $srD$ have not been associated to a software requirement. The result of the evaluation of Equation C.2 shows that $swrA$ has not been connected to a test and that $testB$ and $testD$ are still ‘open’. Both Prolog and SQL should be able to generate this set of faults.

C.2 Prolog

Listing C.1 lists a Prolog file. The Prolog file can be loaded into a Prolog program. The prolog program that is used for demonstration purposes during this project is called SWI-Prolog [50]. Information about the use of the program can be found in its manual [51].

Listing C.1: Example of a Prolog file

```prolog
%% Author: Willem Ridderhof
%% Date: June, 2006
%% Please load this file into SWI-Prolog
```

1
%%% swr  Software Requirement  
%%% sr  Safety Requirement  

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Relations between Safety and Software Requirements
% % run: sr(X), not(relationSR(X,Y)).

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
relationSR(srA, swrA).
relationSR(srA, swrB).
relationSR(srB, swrC).

sr(srA).
sr(srB).
sr(srC).
sr(srD).

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Relations between Software Requirements and Tests
% % run: swr(X), not(relationSWR(X,Y)).

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
relationSWR(swC, testA).
relationSWR(swB, testB).

swr(swA).
swr(swB).
swr(swC).

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Values for the attributes status of the class Test
% % run: value(X,Y), not(Y=ok).

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
value(testA_status, ok).
value(testB_status, open).
value(testC_status, ok).
value(testD_status, open).

attribute(ok).
attribute(open).

% % The rules can be merged in the following way
C.3 SQL

To be able to use SQL, the data has to be transferred to a relational database [15]. Table C.4 shows the data from the example set, but now represented as tables from a relational database.

A SQL query for Equation C.1 has been shown in Listing C.2. This query selects all IDs from the safety requirement table. Then, this table is joined with the relation table shown in Table C.4d. The result is a trace table showing safety against software requirements.

An example trace table has been shown in Table C.5. The table shows that \(srC\) and \(srD\) have no association to a software requirement. In the same way, the associations between software requirements and tests could be analyzed.

Generating a trace table that shows that all test result have the value 'ok' is not necessary while the test table of Table C.4c already has the form of a trace table.
Table C.4: Development data and references represented in the form of tables of a relational database. Primary keys are underlined. When an entity is both a primary and foreign key, that entity has been double underlined.

Listing C.2: SQL query

```
1 SELECT sr.id, relation.consequent
2 FROM (sr
3    LEFT OUTER JOIN relation
4    ON sr.id=relation.antecedent);
```

<table>
<thead>
<tr>
<th>sr.ID</th>
<th>relation.idDeterminant</th>
</tr>
</thead>
<tbody>
<tr>
<td>srA</td>
<td>swrA</td>
</tr>
<tr>
<td>srA</td>
<td>swrB</td>
</tr>
<tr>
<td>srB</td>
<td>swrC</td>
</tr>
</tbody>
</table>

Table C.5: A trace table generated with OCL
C.4 Meta Data Repository (MDR)

A MDR [35] is basically an object oriented database. It is possible to store a model into the MDR. When this model is loaded into the MDR, it becomes possible to store data for the model. This means that instances of the model elements can be created. The model in the MDR describes the format of the data. A model from the MDR has the same functionality as the schema layout of a relational database. The relation between a model and its data has been described by MOF [20]. More about MOF has been written in Section C.4.1

Model Models that can be stored in the MDR are UML models. For instance the IM depicted in Figure 5.5 or a Tool Model of a ToolNet Enabled Tool. There are many different tools to create UML models, among them Rational Rose from IBM [41] and MagicDraw [34]. To ensure that models of all vendors can be stored in the MDR, the MDR requires that a UML model is exported to the XMI format [21]. Files stored in the XMI format can be compared to plain XML files [47], except that the structure of the XMI files conforms to a certain rules. More about XMI can be read in Section C.5.

Data Once a model has been loaded into a MDR, data can be stored, too. Data can be created in two ways. First, it is possible to instantiate classes of the model. Once the classes have been instantiated, attributes can be set. It is also possible to instantiate associations of the UML model. These associations can link two classes to each other. An example of class creation is given in Listing C.3.

Listing C.3: Adding a single data class to the MDR

```java
RefClass class = subPackage.refClass(‘requirement’);
RefObject refObject = class.refCreateInstance( new ArrayList<Object>(0));
```

Second, a whole file with data can be added to the MDR. This method can be used when a vast amount of data has to be added to the MDR. The format of the file with data should be XMI. An example is given in Listing C.4.

Listing C.4: Adding a file with data to the MDR

```java
InputStream input = Utils.getExportFromFile(path_to_data);
MDR.readDataIntoExtent(input, MDR, m_DataPackage);
```

To access the data in the MDR, a specific interface can be used. This is the Java Metadata Interface (JMI). By means of the JMI interface [11], the data from the MDR can be used directly from Java. This is an advantage because ToolNet has
been programmed with Java, too. This means that it is easy to integrate the MDR with ToolNet.

**IM MDR** An example of a MDR is already available in ToolNet. ToolNet’s ToolProxy [17] has been implemented with the help of the MDR. Assume that requirements have been created with DOORS and then, DOORS is closed. From then on, it is not possible anymore for ToolNet to get access to the created requirements. To make the DOORS development objects available even when the tool has been closed, a ToolProxy hook will ask DOORS for an XMI export of its data just before DOORS shuts down. DOORS will return the XMI export and this export is stored in the MDR. Before the data can be stored, a model should be loaded into the MDR. For a DOORS export, DOORS’ Tool Model can be used as such.

Although the ToolProxy can store models and data in an MDR, these models and data are always related to individual tools. To create an integrated view on the data stored in all tools, another approach has to be chosen. The integrated view on all data is modeled in the IM. This IM model can be stored in the MDR, which results in an IM MDR. A special ToolNet routine reads the exports from the tools and stores the data elements and the relations into the IM MDR. A pseudo algorithm for this routine already has been provided by Algorithm 4.1.

**C.4.1 Meta Object Facility (MOF)**

The MOF [20] defines an architecture for describing and defining data and meta data, where meta data describes data. A conceptual overview is given in Figure C.1. The MOF consists of four levels of data abstraction. At the lowest level, level M0, the data of a system resides. For a computer program, the data at this level consists of object instances at runtime. At level M1, the data is described by a model. For the computer program introduced earlier, this can be a UML model describing the system. To create a UML model, a metamodel of UML should be available. Such a metamodel, at level M2, defines concepts as classes, attributes, associations, etc. Finally, concepts of the metamodels are described at level M3. MOF makes it possible to describe a system at various levels of abstraction.

Figure C.2 shows how the IM, Tool Model, development data, and real subsystems can be distributed of the MOF levels. Development objects stored in the various tools resides at level M1. An example of M1 data is a DOORS requirement. DOORS’ Tool Model describes the DOORS data. Therefore, this model is level M2. The IM only defines another view on the data in the development tools. This means that the IM also resides on level M2. The data from the development tools describes the actual system. For example, a DOORS requirement of DaimlerChrysler describes the functionality of a particular subsystem of a car. These subsystems can be interpreted as instances from the data of the development tools. The real subsystems are said to be at level M0.

This thesis has shown how the development data at level M1 can be validated against the IM at level M2.
C.5 The XMI format

If the data in the Safety Argument has to be processed, it is necessary to export the GSN structure. When the Safety Argument is exported, the SAtoOCL component of the Solution Generator can extract the logical equations and convert them to OCL expressions.

There are several different tools to create a Safety Argument. To be able to deal with the data from all of these tools, an exchange format has to be defined. An appropriate exchange format for models is the XML Metadata Interchange (XMI) [21] format. XMI defines a set of rules how a model can be transformed into an Extensible Markup Language (XML) file. The XML file that is generated is XMI conform and is also referred to as XMI file.
We used the ASCE [32] tool to generate our Safety Argument. However, this tool is not able to generate XMI files. This means that the data of the Safety Argument cannot be processed easily. However, the current file format is already a XML format. The only information that is needed to transform a XML export into a XMI export is some metamodel information about the data in of the tool. For an example of a conversion from a XML export into a XMI export with the help of a metamodel, see Freude [17]. He describes such a conversion process for the tool DOORS. The DOORS XML export is converted into an XMI file with the help of a MOF metamodel.

Figure C.3 shows a very simple Safety Argument. By means of this argument an illustration is given what a XMI file looks like. The example XMI file for the given argument is shown in Listing C.5. A XMI file resembles a plain XML file. However, the name of the nodes and the structure of the file are special, so the files can be processed by XMI readers. Please notice that this XMI file is only suitable for demonstration purposes, while it contains an abstraction of a real XMI file. This has been done for the sake of simplicity.

Listing C.5 shows three important nodes, starting at lines 13, 16 and 19. The first node with name Goal represents the top level root goal. This root goal has a sub node, also with name Goal, and this goal represents a sub goal. This sub goal in turn, has a LeafGoal which represents a leaf goal of a Safety Argument. The values of each of these goals can be used by other tools to analyze the safety case tree.

Listing C.5: An example XMI file
The names from the nodes stem from a metamodel. The metamodel describes what a Safety Argument in GSN looks like. A metamodel for the Safety Argument has been given in Section C.5.1.

C.5.1 Meta Model Safety Case Tree

A possible MOF metamodel for the Safety Argument is depicted in Figure C.4. Please notice that this MOF model only models a part of a GSN structure.

The MOF metamodel expresses that a goal can be connected to several other goals, also termed sub goals. At the end, a goal is connected to at least one leaf goal. A leaf goal also is a goal. This leaf goal, in turn, is linked to a solution. A leaf goal can have several types. Currently, only one pattern has been specified, i.e., $\forall aA \rightarrow \exists bB$, where $a$ and $b$ are DevelopmentObjects. This metamodel is extensible because a new leaf goal type can be added easily, once a new pattern has been identified. The MOF metamodel can be used to store the Safety Argument in an XMI file.
Figure C.4: A MOF metamodel for the Safety Argument
Appendix D

Solution Generator

A screenshot of ToolNet with the Solution Generator is depicted in Figure D.1. Please notice that the Solution Generator is termed Safety Analyzer in ToolNet.

![Safety Analyzer as integrated ToolNet component](image)

Figure D.1: The Safety Analyzer as integrated ToolNet component

To start the Safety Analyzer, the white button with caption “SA” has to be pressed. When the Safety Analyzer view has been opened, the user can select and open a data source. A data source represents data from a development tool. When the user clicks on a data source, a menu will be shown that shows analysis options. If the “Analyze” button is pressed, an OCL popup menu as depicted in Figure D.2 will be shown.
The user can choose to enter the OCL constraints directly into a text field. However, when many OCL constraints have to be specified, it might be easier to write these to a file. It is possible to make a reference to the corresponding OCL file.

To use the Solution Generator, some properties have to be configured. The Solution Generator should be configured via ToolNet. A screenshot of the properties popup has been shown in Figure D.3.

In the first text field (“PIM Repository”), the user can specify the path to the IM MDR. The IM has been termed PIM in ToolNet, to emphasize that the IM usually is unique for every system. When a path has been specified, the Solution Generator will not generate a new IM MDR, but rather reuse the existing database stored at the specified location. The IM is only reused if the option of the first drop down menu (“Use Existing Repository”) is set to True. Otherwise, a new repository will be created.

The second text field allows the user to give a pointer to the IM itself. Currently, this model has to be created with Rational Rose and the model should be exported to XMI. The last drop down menu (“Only include selected data sources”) enables the user to analyze only a part of the development data. This option can be used if a large data set of development objects has to be analyzed and the IM MDR cannot be reused because the content is changing too often.
Figure D.3: Configuration of the Safety Analyzer
Appendix E

The Safety Adviser

An important side effect of the automatic generation of a evidence, is that this information can be used for a Safety Adviser, i.e., a tool that informs management on the safety status of a safety related system. Assume that a safety argument with one hundred claims and supportive evidence has been constructed. When the claims have been evaluated, management can be informed about the percentage of claims that hold. Managers are also able to see how many claims do not hold and will be able to give an estimation about the time it costs to work over these claims. The possibility to provide management with traceability information is important for fund raising for the Solution Generator.

E.1 The Project Adviser

The idea of the Safety Adviser is based on the idea of the Project Adviser [13], which is in a very early development stage. The Project Adviser informs management about the maturity of the development artifacts. An example evaluation result generated for an imaginary project is shown in Figure E.1.

The figure shows that a project is decomposed into several sub items. For each item a formula should be defined which maps the maturity of that item to one of the colors red, yellow, or green. Based on this information an estimation about the overall project maturity can be given.

Both Advisers are meant to provide information to managers and developers. At the moment, this data cannot be generated in an automatic way. In the future, managers will be able to make more accurate decisions based on the data provided by the Advisers.

E.2 The Colored Safety Argument

The Safety Analyzer informs management about the safety status of a project. To do this, color information can be used. This idea stems from the Project Analyzer,
which uses red, green, and yellow to give an overview of the project maturity. An example has been shown in Figure E.1.

To color the Safety Argument, the color red and green can be used; red indicates a violation of a constraint and green means that a constraint has been fulfilled. An example Colored Safety Argument has been shown in Figure E.2.

A Colored Safety Argument immediately shows which solutions are violated. The number of violated solutions gives an indication how long it will take to rework the system.

To color the goals of the argument, the argument should be traversed until the leaf goals are being reached. Then the solutions for such a leaf goal can be evaluated. When the result of the evaluation of the solution is known, the solution and the corresponding leaf goal can be colored. When the color of every leaf goal is known, this information can be used to color the remains of the argument. A goal, not being a leaf goal, is red if one of its children is red. When all children of such a goal are green, the goal itself will also be green. Remember that the requisite for calculating the color of a goal is stringent, all children are supposed to be green, before a goal becomes a green color. This is evident, because partial safety is not desired.

E.2.1 Coloring the Leaf Goal

To decide whether a leaf goal is red or green, the leaf goals of the argument should be evaluated. The leaf goals contain a logical expression. Equation B.1, repeated below, shows an expression from the satisfied goal. The valuation, \( v(F) \), of the expression can be either 0 or 1. When \( v(F) = 1 \), the goal containing \( F \) will be
Figure E.2: Colored Safety Argument
colored green. Otherwise, when $v(F) = 0$, the goal will be colored red. This has been shown in Equation E.1.

$$F = \forall aA \rightarrow \exists revRA \land \forall revRA (rev.result == ok)$$

$$leafGoalColor = \begin{cases} 
\text{red} & \text{if } v(F) = 0 \\
\text{green} & \text{if } v(F) = 1 
\end{cases}$$  \hspace{1cm} (E.1)

### E.2.2 Evaluating a Solution

Generating the trace tables based on ToolNet relations and development data can be done as explained in Chapter 4. To generate the argument table, a new OCL pattern should be specified which could look like the pattern depicted in Listing E.1.

Listing E.1: OCL pattern for $\forall revRA (rev.result == ok)$

<table>
<thead>
<tr>
<th>context RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv: self.result = 'ok'</td>
</tr>
</tbody>
</table>

Based on the IM, the OCL engine will be able to iterate over all reviews and verify if the result attribute has been set to 'ok'. The review IDs and a short description can be listed in the table. Furthermore, the attribute value and an evaluation result can be written to the table.

### E.2.3 Coloring a Safety Argument

Coloring a Safety Argument can be done recursively starting at the top level goal. Then all children are processed until leaf goals are being met. To calculate the valuation of the expression of the leaf goal, the solutions should be generated. How to automatically generate a solution has been explained in Chapter 4. When the traceable column of the trace table shows one false entry, then that solution is defined to be false. The same argument holds for the attribute table, if the evaluation column contains one false entry, the solution will be false. When only true values can be found in the respective columns, the solution is said to be true.

When a solution is false, it will be colored red and the corresponding leaf goal will get the valuation $v(F) = 0$. This means that the leaf goal also will be colored red and the parent of the leaf goal will be red too. A pseudo algorithm to calculate the color of the elements of a Safety Argument, has been shown in Listing E.2. The names used in the code are self explaining. Notice that this algorithm only deals with goals and solutions, not with strategy, context, or other GSN symbols.

Listing E.2: Pseudo code to calculate the color of Safety Argument symbols

```java
enum color = {RED=0, GREEN};
color ColorSafetyArgument( Goal );
```
The entry point of the algorithm is the `main` function which should be called with a root goal (which has not been shown). Then the function `ColorSafetyArgument` is called with the root goal. This function initiates the recursion. All children of the
goals will be processed recursively until a leaf goal has been met. The leaf goal will be handled by the \texttt{GetLeafGoalColor} function. A leaf goal will only have solution nodes, so only the \texttt{GetSolutionEvaluation} has to be called. This function will generate various tables (which has not been shown here, see Chapter 4 instead) and return false if one of the table entries is false. Otherwise, true will be returned. Finally, by means of recursion, this color information ripples up to the root goal.