Managing Assurance Cases in Model Based Software Systems

MANAGING ASSURANCE CASES IN MODEL BASED SOFTWARE SYSTEMS

BY
SAHAR KOKALY, M.A.Sc., B.Eng.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

© Copyright by Sahar Kokaly, April 2019
All Rights Reserved

Doctor of Philosophy (2019) McMaster University
(Computing & Software) Hamilton, Ontario, Canada

TITLE: Managing Assurance Cases in Model Based Software Systems

AUTHOR: Sahar Kokaly
M.A.Sc., B.Eng. (Software Engineering)
McMaster University, ON, Canada

SUPERVISOR: Dr. Tom Maibaum and Dr. Marsha Chechik

NUMBER OF PAGES: xvii, 226

To my daughters, Jenna and Lana

"You have brains in your head.
You have feet in your shoes.
You can steer yourself any direction you choose."
— Dr. Seuss, Oh, The Places You'll Go!

Abstract

Software has emerged as a significant part of many domains, including financial service platforms, social networks, medical devices and vehicle control. In critical domains, standards organizations have responded to this by creating regulations to address issues such as safety, security and privacy. In this context, compliance of software with standards has emerged as a key issue. For companies, compliance is a complex and costly goal to achieve and is often accomplished by producing so-called *assurance cases*, which demonstrate that the system indeed satisfies the property imposed by a standard (e.g., safety, security, privacy) by linking evidence to support claims made about the system. However, as systems undergo evolution for a variety of reasons, including fixing bugs, adding functionality or improving system quality, maintaining assurance cases multiplies the effort.

Increasingly, models and model-driven engineering are being used as a means to facilitate communication and collaboration between the stakeholders in the compliance value chain and, further, to introduce automation into regulatory compliance tasks. A complexity problem also exists with the proliferation of software models in model-based software development, and the field of Model Management has emerged to address this challenge. Model Management focuses on a high-level view in which entire models and their relationships (i.e., mappings between models) can be manipulated using specialized operators to achieve useful outcomes. In this thesis, we exploit this connection between model driven engineering and regulatory compliance, and explore how to use Model Management techniques to address software compliance management issues, focusing on assurance case change impact assessment, evolution and reuse. We support the presented approach with tooling and a case study. Although the main contributions of this thesis are not domain specific, for validation, we ground our approaches in the automotive domain and the ISO 26262 standard for functional safety of road vehicles.

Acknowledgements

Every PhD needs a supervisor, and I have been fortunate to have two. First is Dr. Tom Maibaum, who was supportive of the idea that I start a PhD part-time while working as a Research Engineer on the NECSIS project with him. Over the years, Tom and I shared many car rides and travelled on many occasions together. I learned so much from Tom about software engineering, logic and system safety. Tom knows something about everything. He always has the best stories to share, the best food and wine recommendations, and always manages to make me laugh at his puns and jokes. Thank you, Tom, for your supervision, for your support and for your friendship. I am very fortunate to have worked with you and learned from you during the past few years.

Then is Dr. Marsha Chechik, whose co-supervision in this PhD has been invaluable. Marsha welcomed me to her group through our collaboration on NECSIS, and quickly involved me in many of her group's activities. Words cannot express my gratitude to her. She helped me shape my thesis work, taught me how to conduct good research, always answered my questions about various things related to academic life, encouraged me to take on many professional activities, introduced me to many people in the field, and never missed responding to a single email or Skype message from me. Marsha has taught me so much about being a successful academic, and has been a great role model for me both professionally and on a personal level. Thank you, Marsha, for everything. You will be a lifetime friend, and I will always come to you for advice on work, life and parenting.

I am very grateful to have worked with so many amazing people throughout my PhD. My early work was with Zinovy Diskin, who introduced me to Category Theory and taught me a lot about it. I worked very closely with Rick Salay, who taught me so much about academic writing and helped me shape my work. Rick and I had many useful working sessions and conversations, and I am very thankful for his support and friendship. Alessio Di Sandro and Nick Fung were such a pleasure to work with, as well. Alessio led the tool development in *MMINT* and Nick in *MMINT-A*. Thanks to you both for all the work you put into implementation, testing and running experiments needed for this thesis.

I would also like to thank other people I have collaborated with on content presented in this thesis; Mark Lawford, Valentin Cassano, Mehrdad Sabetzadeh, Michalis Famelis and Mike Maksimov. It was a pleasure working with you all.

A PhD journey is not the smoothest of rides and some people can make it so much easier. Richard

Paige has been a wonderful friend and mentor to me and I would like to thank him for that. I have learnt a lot from you, Richard, and I look forward to working with you more in the future. Alan Wassyng, who first taught me in my undergrad, remained a close mentor in my PhD, and I am very grateful to him for his many questions and answers!

I consider myself lucky to have worked in the McMaster Centre for Software Certification (McSCert). Thank you to all my colleagues there for the many useful discussions and for your friendship. In particular, I would like to thank Vera Pantelic for being really supportive and for her listening ears. You are a great friend, Vera. Also, thank you to Lynda, Magda and Laurie for all the administrative help over the years!

It was such a pleasure to have worked with the modeling group at the University of Toronto. I would like to thank everyone in the group for the useful conversations, feedback on my work, and for your friendship. In particular, thanks to Alicia Grubb who encouraged me to keep pushing towards the final stages of the PhD. We did it!

In September 2016, I started a part-time position with General Motors. I have been very fortunate to work with the R&D group in Warren led by Joseph D'Ambrosio. I have learnt so much about model-based engineering and automotive safety working with GM. Thank you Joe, Ramesh, Galen, Sigrid and Lucian for all the useful discussions. Also thanks to Amanda Kalhous for being a supportive manager and a great role model of a successful working woman and mother in the automotive domain.

I would like to thank Mark Lawford and Jacques Carette for accepting to serve on my supervisory committee, for attending all my committee meetings and for the useful comments and feedback that helped improve this thesis. Specifically, I thank Mark for his time and advice on multiple occasions. I would also like to thank my external examiner, Dr. Lionel Briand, for his review of my thesis, attending my defense in person, and raising many interesting points of discussion.

My PhD was funded through the NECSIS project with support from NSERC and General Motors. I was also awarded travel scholarships through CRA-W and SigSoft CAPS, which enabled me to travel to conferences and events. Thank you for the funding!

Last, but not least, comes my family. My parents, Maha and Bishara are the first to thank. They have both sacrificed so much so that my siblings and I can have better lives. Thank you *mama* and *baba* for bringing us to Canada, thank you for encouraging me to start the PhD, and for supporting me through it. You listened to me complain about it numerous times, and kept pushing me and reminding me to keep my eye on the prize. Your love, acceptance, encouragement and vision have gotten me through it. I did this mostly for you, especially you *baba*! I hope I have made you proud.

My siblings, Samer and Rana, have both been my support system. Thank you for listening, for laughing, for helping with the girls, and for just being there. I love you both!

I am so fortunate to come from a big family who loves to talk and be in each others business! My grandparents, aunts, uncles and my cousins, you are all amazing people, and I am so fortunate to have you in my life (nosiness and all!).

I cannot even begin to express my gratitude towards my wonderful husband and life partner,

Majd. You have been through it all with me, the ups and the downs, the good, the bad and the ugly. Thank you *habibi* for everything; for listening, for encouraging me not to quit, for the advice, help and support, for the tea and snacks while writing late at night, for taking full responsibility of our home and family while I travelled to attend conferences, for fixing all my laptop problems, the list goes on and on. I am extremely grateful to you, and I hope this has paid off.

My first daughter Jenna, you lived through my PhD, so it has been like a sibling to you, taking my attention from you at times. You asked me once when you were 5 "*Mama*, when will you stop being a student?", well I think now you have an answer! I love you so much, and I see so much potential in you. Thanks for enduring my stress and work, and thanks for being such a mature and helpful big sister. My baby, Lana, you were the reason I managed to complete this. From the day I found out that I was pregnant with you, I started writing my thesis so I could finish before you were born (of course that didn't happen as planned, but close!). You are my thesis baby, and I am so happy you entered my life during this final year. I love you so much! This thesis is dedicated to you both, and I can't wait to see *all the places you'll go*!

Contents

Abstract iv

Acknowledgements v

I Motivation & Related Work 1

1 Introduction 2
1.1 Motivation 2
1.2 The SafeCar Example 3
1.3 Related Work 5
1.3.1 Model Management 5
1.3.2 Compliance Management Frameworks 6
1.3.3 Languages, Algorithms and Operators for Compliance 6
1.3.4 Modeling Standards, Assurance Cases and Compliance 7
1.4 Gaps Identified In This Thesis 8
1.5 Our Proposal: Model Management for Compliance 10
1.6 Research Questions 14
1.7 Thesis Contributions 15
1.8 Thesis Organization 18

II Megamodel Management 20

2 Background: Model Management 21
2.1 Running Example: Power Sliding Door System 21
2.2 Modeling and Model Management 23
2.2.1 Modeling 23
2.2.2 MDA, MOF, EMF and Ecore 25
2.2.3 Model Management 26
2.2.4 Definitions 27

2.3 Model Slicing . . . 32
2.4 Model Evolution . . . 33
2.5 Chapter Summary . . . 33

3 Background: *MMINT* **36**
3.1 Using *MMINT* for Model Management . . . 38
3.2 *MMINT* Architecture . . . 41
3.3 Chapter Summary . . . 42

4 Megamodel Management with Collection-Based Operators **44**
4.1 Traditional Megamodeling Operators . . . 46
4.2 Megamodel Collection Operators . . . 46
4.2.1 Operator **map** . . . 46
4.2.2 Operator **reduce** . . . 49
4.2.3 Operator **filter** . . . 50
4.3 Application Scenarios . . . 52
4.3.1 Experiment Driver . . . 52
4.3.2 Mass Refactoring . . . 53
4.3.3 Megamodel Transformation . . . 54
4.3.4 Scenario from the Motivating Example . . . 55
4.4 Analysis . . . 56
4.5 Tool Support . . . 57
4.5.1 Using *MMINT* . . . 57
4.5.2 Implementation of Collection Operators . . . 57
4.5.3 Experiments . . . 58
4.6 Related Work . . . 59
4.7 Chapter Summary . . . 60

5 Heterogeneous Megamodel Slicing **61**
5.1 Motivating Example . . . 62
5.2 Megamodel Slicing . . . 62
5.2.1 Slicing algorithm . . . 63
5.2.2 Analysis . . . 64
5.2.3 Discussion . . . 66
5.3 Megamodel Slicing with Collection-Based Operators . . . 67
5.4 PSD Example . . . 69
5.4.1 Megamodels of class and sequence diagrams . . . 69
5.4.2 Slicing of PSD megamodel . . . 70
5.4.3 Post-processing . . . 73
5.5 Related Work . . . 74

5.6 Chapter Summary 76

III Assurance Case Management **77**

6 Background: Assurance **78**
6.1 Software Development in the Automotive Domain 78
6.2 The ISO 26262 Standard 79
6.2.1 ASIL Allocation and Propagation 80
6.2.2 ASIL Decomposition 81
6.2.3 Goal Refinement in ISO 26262 82
6.3 Assurance Cases 83
6.3.1 Modeling Assurance Cases 84
6.3.2 The Goal Structuring Notation (GSN) 85
6.3.3 Claims, Arguments and Evidence (CAE) 86
6.3.4 Structured Assurance Case Metamodel (SACM) 87
6.4 A Survey of Assurance Case Tools 87
6.4.1 Methodology 88
6.4.2 Results 89
6.4.3 Evaluation of the Tools and Discussion 92
6.4.4 Threats to Validity 93
6.4.5 Summary 93
6.5 Chapter Summary 93

7 An Approach for Assurance Case Reuse due to System Evolution **98**
7.1 Introduction 98
7.2 A generic assurance framework for model evolution 100
7.2.1 Objective of Reuse 100
7.2.2 The Framework 101
7.2.3 Additional Model Management Operators 103
7.3 Algorithm Analysis 105
7.3.1 Soundness 105
7.3.2 Relative Efficiency 106
7.3.3 Emergent Properties 106
7.4 Demonstration: PSD example 107
7.4.1 Instantiating the Framework 107
7.4.2 Application to PSD System 107
7.4.3 Evolution of PSD System 108
7.5 Related Work 110
7.6 Chapter Summary 110

8 Instantiating the Approach for Safety, Automotive and GSN **115**
8.1 Introduction . . . 115
8.2 GSN Safety Case Impact Assessment . . . 116
8.2.1 GSN and Annotation Models . . . 116
8.2.2 GSN-IA: GSN Impact Assessment Algorithm . . . 117
8.2.3 Illustration: PSD Example . . . 119
8.3 A More Precise Impact Assessment . . . 119
8.3.1 T1: Increasing the Granularity of Traceability between the System and the Safety Case . . . 122
8.3.2 T2: Identifying Sensitivity of Safety Case to System Changes . . . 122
8.3.3 T3: Understanding Semantics of Strategies . . . 123
8.3.4 T4: Decoupling Revision from Rechecking . . . 124
8.3.5 T5: Strengthened Solutions do not Impact Associated Goals . . . 124
8.3.6 T6: Exploiting Knowledge about ASIL Work-Product Dependencies and ASIL Propagation and Decomposition Rules . . . 125
8.3.7 PSD Example Cost Comparison with T1 . . . 126
8.4 Related Work . . . 126
8.5 Chapter Summary . . . 127

IV Tool Support & Validation **128**

9 Tool Support: *MMINT-A* **129**
9.1 Introduction . . . 129
9.2 *MMINT-A* Requirements . . . 130
9.3 Extensions for *MMINT-A* . . . 130
9.3.1 Assurance Case Metamodel . . . 131
9.3.2 Assurance Case Editor . . . 132
9.3.3 Assurance Case Slicers . . . 134
9.3.4 Change Impact Assessment Algorithm . . . 135
9.4 Power Sliding Door Example . . . 137
9.5 Related Work . . . 138
9.6 Chapter Summary . . . 138

10 Case Study: Lane Management System **141**
10.1 Introduction . . . 141
10.2 The Lane Management System (LMS) . . . 142
10.3 LMS Safety Case . . . 143
10.4 LMS Change Impact Assessment Scenarios . . . 145
10.4.1 Change Scenario 1: Direct System Change . . . 145

10.4.2 Change Scenario 2: Indirect System Change . . . 146
10.4.3 Change Scenario 3: Design Space Exploration . . . 147
10.5 Chapter Summary . . . 148

V Conclusions & Future Work **164**

11 Conclusion **165**
11.1 Summary of Contributions . . . 165
11.2 Future Work . . . 167
11.2.1 Limitations and Improvements . . . 167
11.2.2 Future Research Directions . . . 171

Bibliography **173**

Appendices **185**

A Power Sliding Door Models **186**
A.1 PSD Class Diagram . . . 186
A.2 PSD Sequence Diagram . . . 187

B Lane Management System Models **188**
B.1 LMS Class Diagram . . . 188
B.2 LMS Sequence Diagrams . . . 188
B.2.1 LMS DrivingStraight Sequence Diagram . . . 188
B.2.2 LMS FailureState Sequence Diagram . . . 189
B.2.3 LMS LeftCurve Sequence Diagram . . . 189
B.2.4 LMS SystemOn Sequence Diagram . . . 189
B.3 LMS State Diagrams . . . 189
B.4 Traceability between LMS models . . . 189

C MMINT-A User Manual **198**

List of Tables

3.1 *MMINT* features and where they are illustrated in the scenario. 38
4.1 Experimental results running **map**[`CDMatch`] 59
5.1 Dependency relations for CD and SD slicers. 69
6.1 Tool functionality categories and the corresponding degrees of support. 90
6.2 General tool information. 91
6.3 Evaluation of capabilities of individual tools. 97
8.1 $\texttt{Slice}_{GSN_V}$ dependency rules. 118
8.2 GSN-IA +Ti techniques and improvements. 121
9.1 The assurance case slicer dependency rules. 135
11.1 Summary of compliance management problems from Chapter 1. 167

List of Figures

1.1 A motivating example: SafeCar. 3
1.2 A general model of compliance. 11
1.3 An example of compliance of multiple artifacts to multiple standards. 12
1.4 Assurance case evolution scenario. 16
2.1 Power sliding door system with redundancy [ISO(2011)]. 22
2.2 Power sliding door system class diagram. 22
2.3 Power sliding door system sequence diagram. 23
2.4 The Four-layer Metamodel Hierarchy. Source: [Brambilla *et al.*(2012)]. 24
2.5 Models, metamodels, and platforms. Source: [Mellor *et al.*(2004)]. 25
2.6 Models, mapping functions, and mapping rules. Source: [Mellor *et al.*(2004)]. 25
2.7 Ecore components and their relations. 34
2.8 (Simplified) metamodel for megamodels. 35
2.9 Power sliding door system megamodel. 35
2.10 Metamodel of an mgraph. 35
2.11 An example of a repository including megamodels and a megarel showing the concrete syntax. 35
2.12 Signature of a transformation `CDMerge` for merging class diagrams. 35
3.1 Type megamodel in *MMINT* used for the examples in this chapter. 39
3.2 Screenshot of the final state of MID `Scenario` and selected other MID's used or created in the scenario. 39
3.3 Architecture of *MMINT*. 42
4.1 An illustration of the **union** operator applied to (1) an mgraph of megamodels (megamodel contents shown underneath), and (2) a set of megarels that share the same endpoints. 47
4.2 Signature of the `CDMatch` transformation. 48
4.3 1) An illustration of applying **map** to the `CDMatch` transformation using two input megamodels (megamodel content shown underneath); 2) using the same input megamodel for both arguments. 49
4.4 Algorithm defining behaviour of the **map** operator. 49

4.5 Algorithm defining behaviour of the **reduce** operator. 51
4.6 An illustration of one iteration of **reduce**. First the merge is applied non-deterministically (step 1). Then the relationships to the neighbours of the merged models are computed using appropriate composition operators. Finally, all input elements are deleted. 51
4.7 Algorithm defining behaviour of the **filter** operator. 52
4.8 Experiment driver scenario illustration. 53
4.9 Mass refactoring scenario illustration. 53
4.10 Illustration of transformation signatures for megamodel transformation scenario. (1) Class Diagram (CD) to Entity Relationship (ER) transformation, (2) CD relation to ER relation transformation. 54
4.11 Megamodel transformation scenario illustration. 54
4.12 Motivating example illustration. 55
4.13 Worst case complexity of the operators. 56
4.14 Type megamodel in *MMINT* used for the examples in this chapter. 57
4.15 Screenshot of megamodel for scenario B in Section 4.3 being built in the *MMINT* megamodel editor. 58
5.1 Algorithm for forward megamodel slice. 65
5.2 Signature of polymorphic operators required in the megamodel slicing scenario: (a) Slice; (b) Trace; and (c) SubMerge. 68
5.3 (a) An example megamodel and (b) its sub-megamodel. 68
5.4 Illustration of the megamodel slice scenario. 69
5.5 Slicing criterion S_c[PSD]. 71
5.6 Result of level 1 slicing in 1st iteration. 71
5.7 Result of the 1st iteration. 72
5.8 Result of 2nd iteration. 73
5.9 Output of algorithm after post-processing. 74
6.1 MLOC by product. 79
6.2 Complexity in lines of code. 79
6.3 Development time. 79
6.4 Automotive Software Related Recalls 2011-2016. 80
6.5 The 10 parts of ISO 26262. 81
6.6 ASIL decomposition schemes from ISO 26262. 82
6.7 Methods for software unit testing - ISO 26262 Part 6. 83
6.8 Goal refinement in ISO 26262. 83
6.9 Software development with assurance cases. “P” represents the quality being assured. 84
6.10 Generic assurance case metamodel *AC*. 85
6.11 Core GSN elements from [GSN(2011)]. 86
6.12 Example safety case in GSN from [GSN(2011)]. 94

6.13 CAE element definitions and examples. . . . 95
6.14 An example of a CAE structure. . . . 96
6.15 Overall AC tool support for: a) creation, b) maintenance, c) assessment, d) collaboration, e) reporting and f) integration. . . . 96
7.1 Assurance case evolution scenario. . . . 99
7.2 Algorithm for assessing assurance case impact due to system evolution used in the RMM evolution framework. . . . 104
7.3 Conceptual overview of model management based assurance case evolution framework RMM. Numbers in gray circles correspond to the line numbers of the impact assessment algorithm in Figure 7.2. . . . 104
7.4 Goal tree for system with redundancy. . . . 111
7.5 PSD system without redundancy [ISO(2011)]. . . . 112
7.6 Goal tree after running the evolution algorithm. . . . 113
7.7 Final goal tree for power sliding door system without redundancy. . . . 114
8.1 Fragment of GSN Metamodel extended with validity states. . . . 117
8.2 PSD System Megamodel and Annotation Metamodel. . . . 117
8.3 Algorithm for assessing impact of system changes on a GSN safety case. . . . 118
8.4 Visualization of GSN-IA algorithm. . . . 119
8.5 An annotated GSN safety case for PSD system after running GSN-IA. . . . 120
8.6 Cost equation for effort incurred after an impact assessment. . . . 121
9.1 The AC metamodel in *MMINT-A*. Concrete and abstract classes are distinguished with black and grey borders, respectively. . . . 132
9.2 Screenshot of the AC editor in *MMINT-A*. The main graphical view is on the left, the statistics table upper right, and the impact trace table lower right. . . . 133
9.3 The assurance case CIA workflow in *MMINT* 136
9.4 *MMINT-A* impact assessment algorithm. . . . 137
9.5 Models for the PSD system in *MMINT-A* 137
9.6 The PSD AC after change impact assessment in *MMINT-A*. . . . 139
10.1 LMS system megamodel in *MMINT*. . . . 150
10.2 LMS Hazard Analysis 151
10.3 ASIL determination table from [ISO(2011)] 151
10.4 LMS Safety Case in *MMINT-A* 152
10.5 LMS Safety Case in Astah GSN 153
10.6 LMS Safety Case to LMS CD Traceability Matrix 154
10.7 LMS Annotated Safety Case after Change Scenario 1 155
10.8 Statistics report for Scenario 1 156
10.9 Backward traceability report for Scenario 1 156
10.10 Cost analysis for Scenario 1 156
10.11 LMS Annotated Safety Case after Change Scenario 2 157

10.12 Statistics report for Scenario 2 158
10.13 Backward traceability report for Scenario 2 158
10.14 Cost analysis for Scenario 2 158
10.15 LMS Annotated Safety Case after Change Scenario 3 - LCS TurnOff() 159
10.16 LMS Annotated Safety Case after Change Scenario 3 - LKS TurnOff() 160
10.17 Statistics report for Scenario 3 - LCS TurnOff() 161
10.18 Statistics report for Scenario 3 - LKS TurnOff() 161
10.19 Traceability report for Scenario 3 - LCS TurnOff() 162
10.20 Traceability report for Scenario 3 - LKS TurnOff() 163
10.21 Cost analysis for Scenario 3 - LCS TurnOff() 163
10.22 Cost analysis for Scenario 3 - LKS TurnOff() 163
A.1 Power Sliding Door (PSD) Class Diagram 186
A.2 Power Sliding Door (PSD) Sequence Diagram 187
B.3 Lane Management System (LMS) Class Diagram 190
B.4 Lane Management System (LMS) Sequence Diagram - DrivingStraight 191
B.5 Lane Management System (LMS) Sequence Diagram - FailureState 191
B.6 Lane Management System (LMS) Sequence Diagram - LeftCurve 192
B.7 Lane Management System (LMS) Sequence Diagram - SystemOn 192
B.8 LMS State Diagram 193
B.9 LKS State Diagram 193
B.10 LCS State Diagram 194
B.11 LDWS State Diagram 194
B.12 Traceability between LMS CD and LMS State Diagram 195
B.13 Traceability between LMS CD and LKS State Diagram 195
B.14 Traceability between LMS CD and LDWS State Diagram 196
B.15 Traceability between LMS CD and LCS State Diagram 196
B.16 Traceability between LMS CD and LMS SystemOn Sequence Diagram 196
B.17 Traceability between LMS CD and LMS LeftCurve Sequence Diagram 197
B.18 Traceability between LMS CD and LMS FailureState Sequence Diagram 197
B.19 Traceability between LMS CD and LMS DrivingStraight Sequence Diagram 197

Part I

Motivation & Related Work

Chapter 1

Introduction

1.1 Motivation

From vehicle control to social networks to business transaction platforms, software has come to mediate much of life's activities. In order to protect the best interests of citizens, responsible organizations (e.g., International Organization for Standardization (ISO)) have responded to this trend by creating standards to address issues such as safety, security and privacy. In this environment, compliance of software to standards and regulations has emerged as a key issue. For organizations, compliance is a complex and costly goal to achieve. They may have to comply with multiple standards due to multiple jurisdictions or to address different aspects of the software which may overlap or conflict. The evolution of standards must be tracked and changes assessed. Evidence for claims of compliance must be collected and managed. Finally, maintaining families of related software products further multiplies the effort. Increasingly, models and model-driven engineering are being used as means to facilitate communication and collaboration between the stakeholders in the compliance value chain and further to introduce automation into regulatory compliance tasks. A complexity problem also exists with the proliferation of software models in model-based software development, and the field of Model Management (MM) has emerged to address this challenge. MM focuses on a high-level view in which entire models and their relationships (i.e., mappings between models) can be manipulated using specialized operators to achieve useful outcomes. We exploit this connection between model driven engineering and regulatory compliance, and explore how to use MM techniques to address software compliance management issues.

Although the ideas put forward in this thesis can be generalized beyond safety and the automotive domain, for illustration purposes, we ground our discussion on safety certification and refer to the ISO 26262 standard [ISO(2011)], which addresses functional safety of road vehicles.

Organization. The rest of this chapter is structured as follows: In Section 1.2, we give a compliance example and introduce the problems we want to address. In Section 1.3, we discuss related work and

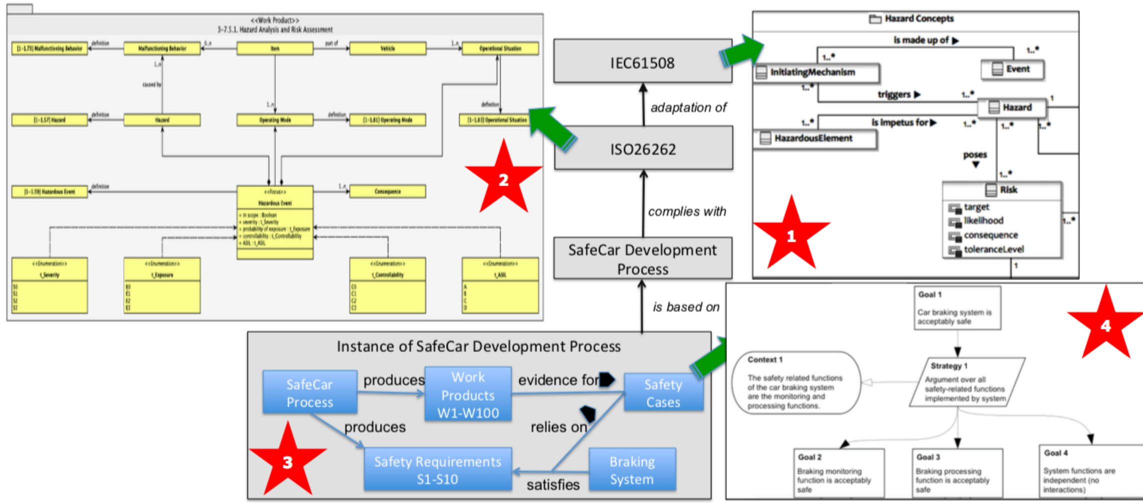


Figure 1.1: A motivating example: SafeCar.

identify gaps that related work does not cover in Section 1.4. In Section 1.5, we demonstrate how the problems stated in the example from Section 1.2 can be addressed using model management. In Section 1.6, we present the research questions that this thesis aims to answer, and in Section 1.7, we present an overview of the thesis contributions. Finally, Section 1.8 explains how the rest of the thesis is organized.

1.2 The SafeCar Example

This section presents an example which we use to motivate the use of model management in the area of regulatory compliance. We present some specific scenarios and generalize them as problems **P1-P6**.

Consider an automotive company SafeCar that produces vehicles. Now suppose that SafeCar either needs to, or would like to, do the following: (1) check the compliance of its braking system safety management lifecycle with the ISO 26262 standard [ISO(2011)] and (2) ensure that its braking system can be certified as being safe. Typically, standards like ISO 26262 are large and take the form of textual documents, and users of the standards often face difficulty answering the questions above [Millett *et al.*(2007)]. ISO 26262 spans 10 parts, including more than 750 clauses, and totaling approximately 450 pages. The size alone makes it difficult to comprehend, thus hindering its application and evaluation. Providing structure to the compliance problem and reducing the complexity and effort of working with standards would provide cost-benefits to these users. Moreover, when dealing with multiple standards and multiple products, SafeCar needs a mechanism to ensure explicit traceability between the various standards and between its products and the standards (**P1: What is a general model of compliance that can be used to define explicit relationships**

between the various artifacts of the compliance activity?).

Figure 1.1[1] illustrates at a high level the relationships between the various artifacts involved in the compliance activity in our motivating example. As seen in the figure, the IEC 61508 standard regulates general functional safety of electrical/electronic safety related systems, and ISO 26262 is an adaptation of IEC 61508. We show what a conceptual model of the Hazard Analysis within a development lifecycle looks like for each of these standards. Refer to the part labelled (1) in the figure for IEC 61508 and the part labelled (2) for ISO 26262. SafeCar will engineer its software development lifecycle (see the part labelled (3) in the figure) to be constrained by the ISO 26262 definition of a software development lifecycle. SafeCar may be interested in demonstrating that its general development process indeed complies with ISO 26262. Moreover, if SafeCar wants to achieve certification that its braking system is safe, its braking system development process will have to identify a set of safety requirements that need to be met. The safety requirements are defined in such a way as to prevent hazards (potential sources of harm caused by malfunctioning behaviour [ISO(2011)]). Safety cases are built to demonstrate that the product meets the safety requirements identified. A safety case may take the form of a GSN Safety Argument [Kelly and Weaver(2004)] as seen by the part labelled (4) in the figure.

But compliance is not necessarily a one-time type of activity; if SafeCar knows that a product complies with ISO 26262, and ISO 26262 undergoes a revision, SafeCar would like to have a means of checking compliance to the new version of ISO 26262 without starting from scratch. Ideally, they would like to reuse as much of the compliance artifacts (arguments, claims and evidence) used to demonstrate compliance to the older version of the standard as possible. Similarly, if the product they have already certified itself evolves to a newer version, they would like to reuse as much of the compliance artifacts from the previous version as possible (**P2: How can claims and evidence be reused due to standard or product evolution?**). In some cases, a company like SafeCar may choose to demonstrate partial compliance to a standard; this could be due to business decisions to minimize the cost of compliance, or perhaps to address only a part of the standard that is of interest (e.g., Hazard Analysis and Risk Assessment). Similarly, the company may choose to address compliance of a part of its system to a standard (e.g., only parts that interact with the braking system.) (**P3: How do we extract relevant parts of a standard or a system for checking compliance?**).

Furthermore, SafeCar may be interested in checking that a product (e.g., its infotainment system that runs on its vehicles) complies with multiple standards (e.g., a privacy standard and a security standard) and would like to achieve this in a one-step compliance check (**P4: How can we reduce the effort of complying to multiple standards?**).

As products at SafeCar may belong to a product line, SafeCar may be interested in ensuring that the entire product line complies with a standard (**P5: How do we define a way to lift compliance from single products to product lines?**).

[1] IEC65108 Hazard concept model from [Panesar-Walawege *et al.*(2013)], ISO 26262 Hazards and Risk Assessment model from [Cassano *et al.*(2016)] and GSN Safety Argument from slides based on [Hawkins *et al.*(2015)].

On the other hand, regulatory bodies may be interested in comparing their standards (e.g., ISO 26262 for automotive) with standards in a different domain (e.g., RTCA DO-178B for avionics) to understand the commonalities and the differences (**P6: Can we identify relationships between standards?**).

This section has presented an example to motivate the use of model management in the area of regulatory compliance. Next, we will review related work in this area in order to identify some gaps that this thesis will address.

1.3 Related Work

In this section, we first briefly review work related to model management which we will refer to in Section 1.5, and then work related to the use of model management for compliance related problems.

1.3.1 Model Management

In the field of model-driven software development [Beydeda *et al.*(2005)], a complexity problem exists due to the proliferation of software models. As such, the area of Model Management [Bernstein(2003)] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their *relationships* (i.e., mappings between models) can be manipulated using *operators* (i.e., specialized model transformations) to achieve useful outcomes. To help visualize and manipulate collections of models and their relationships, model management uses a special type of model called a *megamodel* [Diskin *et al.*(2013)] in which the elements represent models and the links between the elements represent relationships between the models. For example, Figure 1.1 is a megamodel. Model management operators that have been studied include the following:

- The *slice* [Nejati *et al.*(2012)] operator accepts a model and a *slicing criterion* and extracts the subset of the model satisfying the criterion. Model slicing is a way to manage model complexity by focusing on a relevant subset of a model.
- The *match* [Bernstein(2003)] operator accepts two models and produces a relationship containing mappings between equivalent (or similar) elements in the models. This is usually interpreted as identifying the overlap between the models.
- The *diff* [Bernstein(2003)] operator accepts two models and produces a model that represents the differences between the models. Model differencing aids the comparison of model content, e.g., across different versions.
- The *merge* [Brunet *et al.*(2006)] operator accepts two models and a relationship expressing the overlap between them and produces a model that combines the content of the models according to the overlap. Model merge must address the issue of conflicts that could occur when the content is combined.
- The *lift* [Salay *et al.*(2014)] operator accepts a model transformation and produces a product

line transformation that behaves the same way as the original model transformation for each product in the product line. Transformation lifting saves effort by allowing model transformation to be reused for product lines of models.

• The *filter* [Salay *et al.*(2015)] operator accepts a megamodel and a model (relationship) property and produces a megamodel with all models (relationships) not satisfying the property removed. Filtering a megamodel is useful for managing the complexity of large collections of related models.

• The *map* [Salay *et al.*(2015)] operator accepts a megamodel and a model transformation to produce the megamodel that results from applying the transformation to all applicable models and relationships in the input megamodel. Mapping a transformation over a megamodel is used to reduce the effort of applying a transformation to the elements of a large collection of related models.

Each of these operators can be viewed as an *abstract* transformation that defines a class of concrete transformations, i.e, the implementations that refine the operator for particular model types. For example, a model merge of class diagrams is implemented differently than a model merge of state machines. Another widely used class of transformations used in model management are *bidirectional transformations* [Diskin *et al.*(2010)]. Bidirectional transformations are used to keep two related models synchronized when one of the models changes (e.g., via model co-evolution, correction, etc.) by generating the update for the other model.

We are not aware of any existing research specifically on applying model management techniques to assurance case management; however, there is substantial research that is related to using model-based approaches to aid in the compliance problem. We consider this research to be complementary to our objectives and review it here.

1.3.2 Compliance Management Frameworks

Several compliance management frameworks have been proposed in the literature. [Hamou-Lhadj and Hamou-Lhadj(2007)] sketches a proposal for comprehensive compliance management in software organizations. The authors discuss the need for tool support for working on large, possibly overlapping or conflicting compliance documents. Researchers have proposed generic metamodels that can be used to structure any safety standard as well as the related safety case information in a project [de la Vara and Panesar-Walawege(2013), Habli and Kelly(2008)]. An advantage of such an approach would be the possibility of providing generic tooling to address the compliance to any safety standard.

1.3.3 Languages, Algorithms and Operators for Compliance

Specific algorithms for different aspects of compliance management have been proposed. For example, [Nejati *et al.*(2012)] developed a model slicing algorithm for extracting parts of a design that are relevant to a given safety requirement; the survey [Ghanavati *et al.*(2011)] identifies research strands

proposing methods for the extraction of requirements models from regulation documents; and, the authors of [Ghanavati *et al.*(2014)] propose an algorithm for comparing multiple regulations.

Argumentation modeling languages have been studied extensively as the basis for expressing compliance claims. These include the Goal Structured Notation(GSN) [Kelly and Weaver(2004)], Claim, Arguments and Evidence (CAE) [Emmet and Cleland(2002)] and OMG's Structured Assurance Case Metamodel (SACM) [2]. Of these, SACM represents the latest in the evolution of these notations and is also proposed as a standard. We aim to use modeling languages such as these to express compliance relationships.

1.3.4 Modeling Standards, Assurance Cases and Compliance

Standards and regulations can be expressed as models. For example, [Luo *et al.*(2013)] show that the ISO 26262 standard for functional safety of road vehicles can be represented by a combination of a structure model, conceptual model, process model while [Panesar-Walawege *et al.*(2013)] proposes a conceptual model of IEC 61508. We consider this work as a prerequisite for applying model management techniques to the compliance problem.

A recent survey [Nair *et al.*(2015b)] regarding how industry addresses evidence management for compliance to safety standards, reveals that much of evidence management is done manually with little reliance on advanced tool support. This is also reflected in the survey done in [de la Vara *et al.*(2016a)]. However, key issues such as evidence traceability, structuring evidence as models and assessing evidence completeness or correctness correspond directly to general modeling issues: expressing model relationships (including traceability relationships), structuring model content using metamodels and checking model completeness/correctness. We may conclude that adapting automated or semi-automated techniques from model management to address these issues can benefit the state of the practice.

As mentioned in Section 1.3.3, modeling notations for assurance cases have been proposed, most notabley the Goal Structured Notation(GSN) [Kelly and Weaver(2004)]. These are presented in more detail in a dedicated section in Chapter 6.

Model-based approaches for compliance have also been studied. For example, the authors of [Habli *et al.*(2010)] propose model-based assurance for justifying automotive functional safety. In their work, they address one component of the overall safety case, namely the assurance of the functional safety concept. In particular, they examine how model-driven development and assessment can provide a basis for the systematic generation of functional safety requirements.

The authors of [Gallina(2014)] propose a model-driven safety certification method for process compliance. They focus on safety processes mandated by prescriptive standards and identify process-related structures from which process-based arguments can be generated and more easily reused.

In [Conrad *et al.*(2012)], an artifact-centric compliance approach for ISO 26262 projects using model-based design is proposed. The approach is intended to streamline ISO 26262 compliance

[2]OMG: `http://www.omg.org/spec/SACM/1.1/`

documentation for software developed using model to code generation.

The authors of [Stürmer *et al.*(2012)] consider reviewing software models in compliance with ISO 26262 and introduce best practices for model reviews with the aim of ensuring safety-related objectives and adherence to the standard.

In [de la Vara *et al.*(2016b)], a model-based specification approach of safety compliance needs for critical systems is proposed by introducing a holistic generic metamodel. The metamodel abstracts concepts and criteria from different safety standards, and its application results in models for structuring and managing compliance information. The authors claim that the proposed metamodel can be used for most critical computer-based and software-intensive systems.

The line of work by Yaping Luo and Mark van den Brand has also considered the use of a model based approach to compliance. For example, in [Luo *et al.*(2014)], they propose to use conceptual models in the form of metamodels to support certification data reuse and facilitate safety compliance. In [Luo *et al.*(2016)], a categorization of GSN-based safety cases and patterns is proposed with the aim of using it to identify similar safety cases or patterns and facilitate safety case reuse. Their work in [Luo *et al.*(2015a)] offers a modeling approach to support safety assurance in the automotive domain by proposing a rule-based approach that enables extracting a conceptual model from safety standards or project guidelines. Finally, in [Luo *et al.*(2017)] they study safety case assessment and propose an approach and tool support for it based on evidential reasoning.

Some work has focused on the notion of an assurance case template. In [Chowdhury *et al.*(2017)], the authors motivate the use of assurance case templates as a means of capturing the requirements of a complex standard such as ISO 26262. They present eight principles that can be used to drive the transformation of clauses in the standard into claims and evidence in the assurance case template.

1.4 Gaps Identified In This Thesis

We have identified two major gaps in the literature: the first related to the management of collections of models and relationships between them, and tool support for it, and the second related to managing assurance cases in heterogenous model-based systems, particularly due to evolution. We discuss these gaps in more detail in this section.

A major challenge for the practical application of model-driven engineering is the difficulty of managing a large collection of heterogenous inter-related models. MDE inevitably involves construction, integration, and maintenance of a large number of different models, representing different versions over time, different variants across a product family, different options for implementation, and so on. The goal of model management is to provide a systematic way to represent the relationships between such a set of models, and a set of tools for manipulating inter-related models. Such tools include comparing similar models, identifying dependencies, inconsistency checking, propagating change, tracing design decisions, and merging partial models. Many of these issues have been studied and solutions to them have been proposed in the modeling community. However, what is missing is a systematic approach for representing relationships between a large set of heterogenous

models, a means to visualize these relationships, and tool support for manipulating them. This represents the first gap this thesis aims to address as a means for filling the second gap regarding assurance case management in heterogenous model-based systems.

Safety engineers in various domains, including automotive, experience difficulties with safety case maintenance. As stated in [Kelly and McDermid(2001b)], the main reason for this is that they do not have a systematic approach by which to examine the impact of change on a safety argument. The authors of [Borg *et al.*(2016)] performed a study which suggested that engineers spend 50-100 hours on Change Impact Assessment (CIA) per year on average. The second most commonly mentioned CIA challenge is related to information overload. The three most senior engineers in the study reported that obtaining a system understanding is hard due to the complexity of the systems. The sheer number of software artifacts involved makes traceability information highly complex. Based on the results of [Borg *et al.*(2016)], determining how a change impacts the product source code seems to be less of a challenge than determining impact on non-code artifacts, e.g., requirements, specifications, and test cases. In [de la Vara *et al.*(2016a)] and [Leveson(2011)], the authors further discuss the problem of CIA being a challenge in safety-critical systems. Specifically, Leveson [Leveson(2011)] mentions that inadequate CIA has been among the causes of accidents in the past. Thus, the current state of practice can clearly benefit from improved CIA techniques, especially to help perform safety assurance more cost-effectively.

Although related work has used model-based approaches to either construct or assess assurance cases, it has not provided means for automating parts of the assurance case management process, particulary due to change. This observation is further supported by the following two recent surveys.

First, the survey done in [de la Vara *et al.*(2016a)] aimed to provide new insights into how safety evidence change impact analysis is addressed in practice. Based on the survey, it was identified that changes during system development, system modification and re-certification, and component reuse are examples of situations in which safety evidence change impact analysis can be necessary. The survey showed that given the difficulty in cost effectively managing re-certification (either due to system, domain or standard changes), research efforts targeted at this problem are necessary. The authors reported the lack of change impact analysis tools for safety artifacts as a most frequent challenge and concluded that the evolution of safety cases should be better managed and the level of automation in safety evidence change impact analysis is low.

Second, the survey done in [Cheng *et al.*(2018)] attempts to understand how practitioners perceive assurance cases in safety-critical software systems. Again, one of the challenges the survey identified is the lack of tool support for working with safety assurance cases. The study also showed that since changes in software, especially in requirements, can result in changes in safety assurance cases, effectively managing change is a challenge that practitioners faced. The lack of tool support also contributed to this challenge.

Based on the results of these two surveys, as well as our review of related work, we have identified a need for work on assurance case management, specifically change impact analysis and tool support. This thesis aims to address these gaps by providing a novel model-based approach to compliance

management, focusing on a technique for automating assurance case change impact analysis through model based operators and workflows, as well as tool support for it.

1.5 Our Proposal: Model Management for Compliance

Since standards and regulations can be expressed as models (e.g., [Luo *et al.*(2013)]), and compliance claims and links to evidence can also be captured in a model [Ghanavati *et al.*(2011)], the modeling community is starting to work in the direction of providing support for compliance management. Specifically, the Object Management Group has recently issued a modelling language standard called the Structured Assurance Case Metamodel (SACM) [OMG(2015)].

There is evidence that the MM approach to compliance management may fill the gaps left by other research on this topic. For example, a survey [Abdullah *et al.*(2010)] comparing international research in compliance management with the compliance needs of the Australian industry found that a key area where support is needed is in managing the impact of regulations. Specifically, they identified four factors: (1) frequent changes to regulations; (2) legislation weaknesses; (3) inconsistencies; and, (4) overlap in regulations. Furthermore, they identified a disproportionate lack of research in these areas as compared to other issues in compliance management. Although they hypothesized that this may be due to the fact that these factors may be in the legal (as opposed to IT) realm, all but factor (2) correspond to general modelling problems that have been addressed with model management techniques. For example, (1) corresponds to the change propagation due to model evolution, (3) to addressing inconsistencies, which is a standard step within a model merge operation, and (4) to identifying overlaps, which is the outcome of the model match operation.

In this section, we take advantage of the above, and demonstrate our ideas for using MM techniques in the area of compliance. We refer to the problems identified in our motivating example in Section 1.2 and propose general solutions to them.

P1: Creating a general model of compliance that defines explicit relationships between the compliance artifacts. We illustrate our general model of compliance in Figure 1.2. The idea is that regulators (such as the ISO organization) define standards (such as ISO 26262), and these standards are defined as a collection of interrelated models. Each model addresses a different view (process, concepts, etc.) of the standard. In model management terminology, a megamodel [Diskin *et al.*(2013)] represents a model of models and relationships between them. A standard can then be seen as a megamodel in this sense, but for simplicity, we illustrate it as a single model and refer to it as the Standard Model (SM). A company would ideally also have a model of its development process, which again can be seen as a megamodel linking different views of the company's software development process. Again, for simplicity, we will illustrate it as a single model and refer to it as the Software Development Process Model (SDPM). The SDPM should be engineered to satisfy constraints in the SM; therefore, compliance with the SM can be shown via an Assurance Case (AC) that is defined as a relationship between SM and SDPM.

The SDPM can undergo refinements within the company and can be tailored with respect to

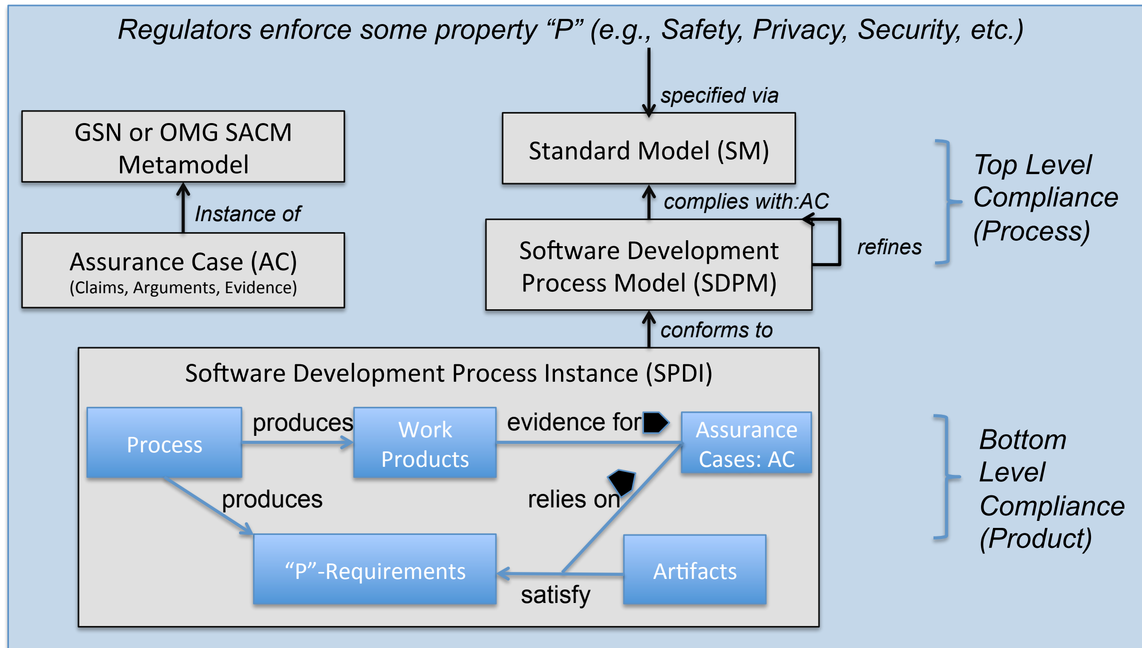


Figure 1.2: A general model of compliance.

the product it will be used in producing. However, as soon as the company creates an instance of its SDPM for a certain product, this instance should conform to the metamodel defined by the SDPM. For example, concepts and processes in the instance should conform to their respective definitions in the SDPM. Depending on the property "P" that is to be met (Safety, Privacy, Security, etc.), "P"-Requirements are produced as part of the instance process, along with the various work products that are defined in the standard. When preparing to certify an artifact to meet the property "P", assurance cases (or "P"-Cases) are provided to support this. The Assurance Case (AC) is constructed using evidence (given by the work products), and can be seen as an instance of a GSN [Kelly and Weaver(2004)] or an OMG SACM [OMG(2015)] metamodel which defines how to construct these instances.

Figure 1.2 depicts this model via two levels of compliance. The top-level (or *process*) compliance is given by an assurance case, either stating that an item complies or providing rationale for non-compliance. The bottom-level (or *product*) compliance is given by a "P"-case that demonstrates that the product satisfies the property "P" as specified in the standard. In the UML terminology, Figure 1.1, which describes a concrete example, can be seen as an Object Diagram that is an instance of the Class Diagram given by our general model in Figure 1.2.

This general model of compliance need not be limited to a single standard and a single product. For example, Figure 1.3 shows a model with several products – a car's braking system and its infotainment system – complying to multiple standards – safety, security and privacy.

Note that the model in Figure 1.2 is itself a megamodel: the models represent the standards,

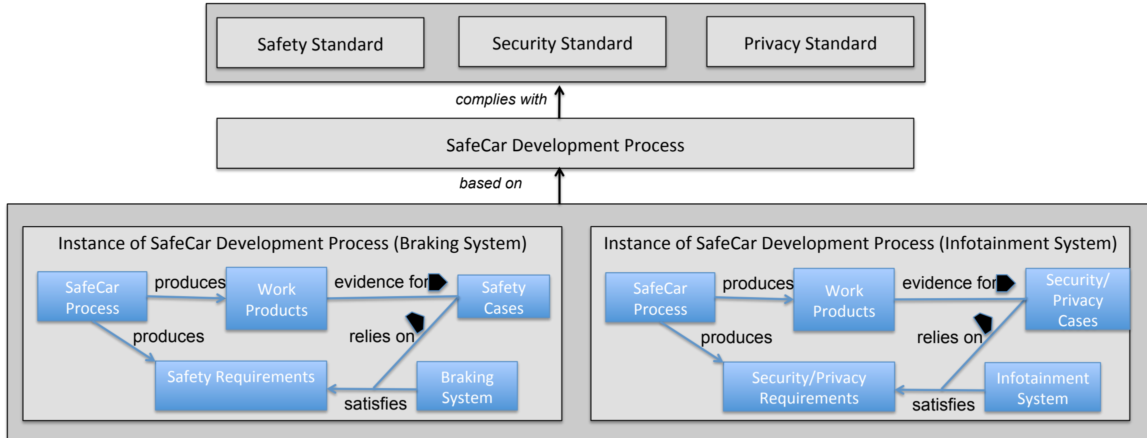


Figure 1.3: An example of compliance of multiple artifacts to multiple standards.

processes and artifacts, and the relationships between them are made explicit. Some examples of these relationships are: *specialization* (of a standard from a general domain to a more specific one), *compliance* (of a process to a standard), *refinement* or *evolution* (of a standard) and *instantiation* (as in the case of the instance of the SDPM). This observation opens up the opportunity of using collection-based operators on megamodels of standards. One possible scenario is: given a collection of standards, use the *filter* operator to obtain "relevant ones" based on some criteria. Perhaps we are interested in the process-related standards, so we filter out all the ones with type "activity diagram". Also, given a set of standards with a shared ontology, we can use the *map* operator to translate some concept in the ontology to another (e.g., "automobile" to "vehicle").

P2: Reusing evidence and other assurance artifacts due to standard or product evolution. Evolution of a standard involves a transformation of the standard model from an old version to a newer one. But since there might have been some products that were already shown to comply with the standard, its evolution means that the compliance of these products needs to be re-evaluated. More concretely, in Figure 1.1, suppose the ISO 26262 standard undergoes a revision (which it currently does), leading to the introduction of a new attribute, say "cost", in the "Risk" class. This means that the SafeCar Development Process has to evolve to ensure that it complies with the newer version of the standard. This problem is commonly addressed in model management using a *bidirectional model transformation*. The main idea is that a *match* operator is used to establish links between the old and new versions of ISO 26262, and a *diff* operator is used to show what new information has been added. Then, the SafeCar Development Process is transformed to a new version that is compliant with the new version of ISO 26262. The challenge added here is in managing the assurance case artifacts (claims, arguments and evidence) that are attached to the compliance relationship. Not only do these need to be re-evaluated, but it would be very cost-effective if these artifacts could be reused as much as possible. Evolution of a product (e.g., the braking system in Figure 1.1) that is known to comply with a standard also means that compliance needs to be

re-evaluated. Again, this can be expressed via a *bidirectional model transformation* definition which also needs to be adapted to take into account the assurance case artifacts and reuse them as much as possible to minimize the cost of compliance.

P3: Extracting relevant parts of a standard or a system for checking compliance. Based on some criteria of interest (*slicing criteria*), extracting only relevant parts of a standard (e.g., those related to hazards and risks in ISO 26262) can be achieved with the use of the *slice* operator. This could also apply to the product undergoing compliance; the slicing criteria could be chosen to select parts of the model related to the braking system for example. Both scenarios (standard and product slicing) could arise due to business decisions to demonstrate partial compliance with a standard in order to narrow the focus to the compliance tasks that are feasible to perform given the stage of development that the system is at.

P4: Effort reduction of compliance to multiple standards. As demonstrated in Figure 1.3, we may wish to check the compliance of a system (e.g., infotainment) to multiple standards (e.g., security and privacy). Some concepts presented in these standards may overlap (e.g., the notion of a "threat"). An MM operator *match* can be used to help identify these overlaps. Once their overlap is defined (note that this can be empty), standards can be merged with the *merge* operator creating a merged standard with traceability mappings back to the original standards. These mappings are important to have, especially when the original standards evolve, which can be reflected in the merged standard (via model co-evolution). Compliance of the infotaintment system in Figure 1.3 to the security and privacy standards can therefore be reduced to a single compliance checking problem of the system to the merged standard. As in model management, a challenge here is actually creating the overlaps between the standards which involves a high degree of human input. Another challenge is understanding how to provide assurance case artifacts when demonstrating compliance to the merged standard (e.g., how do these artifacts relate to the artifacts we would get when demonstrating compliance to each of the individual standards.).

P5: Lifting compliance from a single product to a product line. Products are often modified and reused in new applications, forcing companies to develop and maintain product lines. Product line safety is a natural important requirement of this activity [Palin *et al.*(2011)]. Lifting the problem of compliance of a single piece of software to a single standard to that of an entire software product line can be achieved using the model management *lift* operator. Once again, the challenge here is understanding how the assurance artifacts are expressed in the lifted version of the compliance relationship.

P6: Identifying relationships between standards. In order to create a common certification framework which spans different vertical markets, e.g., in the transport sector (railway, avionics and automotive industries), and facilitates the reuse of assurance assets within, across, and between domains, creating mappings between these standards is an important step. The *match* operator can be used to identify these overlaps, in the simplest case using a "name-match" criterion which links elements with matching names. Creating these overlaps is also an essential step in performing

other useful operations such as *merge* and for maintaining consistency between standards (model synchronization). Another relationship that one may be interested in identifying is representing the differences between two standards (these could be two versions of the same standard). This could be seen as differences in concepts (e.g., what appears in one and not in the other), and can be achieved using the model management *diff* operator.

Companies like SafeCar could benefit from the automation of parts of the compliance activities. For example, we have identified the following in Part 2 of the ISO 26262 standard:

"The Safety Case has to be reviewed for completeness:

C.2 Review of the completeness of the safety case (see Section 6.5.3 in ISO 26262)

C.2.1 Confirmation that the work products referenced in the safety case are available and sufficiently complete, so that the item's achievement of functional safety can be adequately evaluated.

C.2.2 Confirmation that the work products referenced in the safety case: are traceable from one to another, have no contradictions within or between work products, and either have no open issues that can lead to the violation of a safety goal, or have only open issues that are controlled and have a plan for closure."

Once the standards and compliance artifacts are structured in a model management framework, various techniques can be used to analyze and verify these models. In the ISO 26262 example, activities like checking the completeness of a safety case, checking the availability (existence) and the completeness of work products, checking traceability between work products, checking for contradictions (conflicts) can certainly be at least semi-automated given appropriate models that are supported by expert judgement.

Another general advantage of using a model management framework for managing standards and compliance artifacts is the use of generic *model transformations*. Model transformations could be useful for scenarios such as providing different views (models) of the same standard. This could aid in improving communication and comprehensibility of the standard. Another useful application for model transformations could be when companies decide to adopt regulations and best practices from a different country/jurisdiction, and therefore standards can be translated to fit the company's needs.

1.6 Research Questions

This thesis will address some of the problems presented in Section 1.5. We will focus primarily on **P1** and **P2** in order to show the applicability and effectiveness of model management for regulatory compliance problems. Specifically, the thesis will provide a method for assessing the impact of system design changes on its assurance case, identifying which parts can be reused and which ones need to be revised or rechecked. This will be done via the introduction of adapted model management operations and workflows for assurance cases, Also, given the model management interactive tool MMINT [Di Sandro *et al.*(2015)], we implement some of the adapted model management operations and workflows for assurance case management. We also validate our approach and tool on a case study from the automotive domain.

Hypothesis. *The state of practice in assurance case management can be made more rigorous and efficient using model management techniques.*

The goal of this work is to demonstrate the effectiveness of applying model management techniques in the area of regulatory compliance. We address this by answering the following set of research questions:

RQ0: Managing Heterogenous Megamodels. How do we formally define heterogenous collections of models and relationships between them? Can we define operators to manage such collections and workflows to implement certain scenarios of interest over them? Can we provide tool support for all of this?

RQ1: Assurance Case Modeling. How do we best develop an assurance case metamodel that captures the basic assurance artifacts and relationships and dependencies between them? What are the assumptions and constraints on this metamodel?

RQ2: Modeling the Compliance Ecosystem. How do we best model the compliance ecosystem including system design, standards, system development processes and assurance cases? What types of relationships (e.g., mappings, refinements, etc.) exist between these entities?

RQ3: Assurance Case Operators. What type of traditional model management operators (e.g., match, merge, slice, etc.) can be used in the context of assurance cases? How can these operators be adapted to work with assurance cases, taking into account the elements of an assurance case (claims, arguments, evidence), dependencies between them, and inference rules used to relate them?

RQ4: Model Management Workflows for Compliance Scenarios. What are some compliance management scenarios where model management workflows can be implemented to address them? What are these model management workflows and can they be made generic? If they are semi-automated and require human intervention (expert opinion) to complete, how can we best involve experts in completing them?

RQ5: Application in the Automotive Domain. What are the kinds of processes, standards and constraints we have to work with in order to apply our solutions in the automotive domain? How do we adapt our assurance case metamodel, operators and workflows to work in this context?

RQ6: Tool Building. How do we best provide tool support for our approaches? How can the compliance management workflows we present be implemented on top of an exisitng model management tool? What type of adapatation is needed? What are the missing components that will allow us to work with and manage compliance ecosystems which are heterogenous in nature?

RQ7: Validation. Given the tool support, how do we validate the effectiveness of the approaches presented as model management workflows in **RQ4**? How do we evaluate them with respect to improving efficiency?

1.7 Thesis Contributions

This section details the contributions of the thesis.

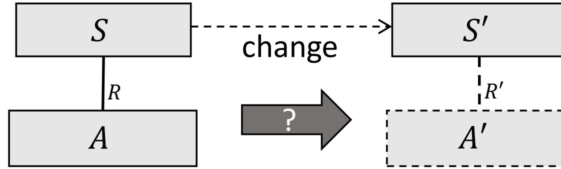


Figure 1.4: Assurance case evolution scenario.

- **Megamodelling: Foundations, Operators, Workflows and Tool Support.** A *Megamodel* is an entity that captures collections of models and relationships between them. As this forms the underlying formal model of the various components in a compliance ecosystem (requirements, system design, standards, development processes, assurance cases, etc.), we did preliminary work to give it a formal treatment, defined collection-based operators (namely, Filter, Map and Reduce) to enrich megamodel management, presented an approach for slicing of heterogenous megamodels to aid in model evolution, and discussed our tool support (via *MMINT*) for it in Chapters 3, 4 and 5, as part of addressing **RQ0**.

- **Model Management for Regulatory Compliance.** There is evidence that the model management approach to compliance management may fill the gaps left by other research on this topic [Abdullah *et al.*(2010)]. In Section 1.5, we took advantage of this and showed our ideas for using model management in the context of compliance. We addressed **RQ2** by presenting a general model of compliance which encapsulates the various artifacts and relationships between them. This model can be further improved to include additional artifacts and relations. Section 1.5 also addressed **RQ4** by identifying compliance management scenarios that can be mapped to model management solutions.

- **Assurance Case Change Impact Assessment due to System Evolution.** In Chapter 7, we focus on one of the scenarios we presented in Section 1.5 – assurance case reuse due to system evolution – and develop it in detail. Figure 1.4 illustrates the scenario at a high level. Assume that S describes the specification for the software in a vehicle. In addition, a type of assurance case A, called a *safety* case, has been developed complying with the ISO 26262 vehicle functional safety standard [ISO(2011)]. Safety case A contains perhaps thousands of safety claims about different components of the vehicle, as well as arguments and evidence to support these claims. Now if S is evolved to S' – for example, as a result of a new requirement or a bug fix – a corresponding safety case A' for S' must be developed. Due to the complexity and effort required to develop a safety case, there is strong incentive to reuse as much of A as possible in the creation of A'. We address this problem using a model management strategy by developing an assurance case impact assessment algorithm. The algorithm does the following: given a model of the system and an assurance case linked to it, and a known change to the system design, the algorithm uses a series of model management operations (e.g., slice, merge, trace) to assess the impact of these changes on the assurance case. It also uses the

dependencies between the elements of the assurance case to propogate this impact across the assurance case itself. The algorithm eventually outputs an annotated version of the assurance case, which highlights the elements that can be reused (unaffected by changes), should be rechecked (indirectly affected by the changes), or need to be revised (directly affected by the changes). In doing so, we further address **RQ4** by providing a model management workflow that performs impact assessment on an assurance case due to system evolution – one possible compliance management scenario.

- **Assurance Case Management.** In order to fully realize our approach in Chapter 7, we define a complete assurance case metamodel that captures the entities and dependencies in a way that allows us to reason soundly about them in Chapter 8. This addresses **RQ1**. We also specified slicing operators for this metamodel, which enable us to manage assurance cases in a model management fashion under various compliance scenarios addressing **RQ3**.

- **Application in the Automotive Domain.** To help understand the validity of our approaches, we ground them in the automotive domain, and work with the ISO 26262 standard for functional safety of road vehicles. To do so, certain aspects of the standard (e.g., the notions of ASIL integrity levels and Work Products) have been captured in our assurance case metamodel in Chapter 8. This allows us to trace the assurance case to both the standard and the system design, which enables a more effective assurance case impact assessment under different change scenarios. In doing so, we address **RQ5** and parts of **RQ7**, especially those related to assessing the applicability of our approach.

- **Tool Support and Evaluation.** For the purpose of addressing **RQ6**, we have extended the MMINT [Di Sandro *et al.*(2015)] framework to include assurance cases in Chapter 9. We add the capability to work with heterogenous megamodels (which represent the compliance ecosystem), and operators that can be applied to them. We implement a workflow language that allows us to combine model management operators to achieve certain scenarios of interest. Next, we incorportate assurance cases in the tool by adding the assurance case metamodel as a type and implementing its type-specific model management operators (e.g., slice). We also implement the model management workflow we presented for assurance case impact assessment using our workflow language. In Chapter 10, we report on a case study from the automotive domain which serves as a validation of our approach and addresses the remainder of **RQ7**. Finally, *effectiveness* of our approach can be linked to cost savings, both when assessing compliance (by adding automation) and when re-assessing compliance under incrementality and evolution (by enabling reuse of assurance artifacts). We evaluate that as part of the case study in Chapter 10.

1.8 Thesis Organization

The thesis is organized into five main parts, the first one being this introduction. The rest of the thesis is organized as follows:

Part II. This part focuses on megamodel management foundations, without going into the application to assurance case management. It is organized as follows:

- In Chapter 2, we introduce a running example, as well as relevant background literature, concepts and notations related to modeling, megamodels, and model and megamodel management.
- In Chapter 3, we introduce *MMINT*– a tool for interactive model management, which is used as a tool for evaluation of our megamodeling approaches.
- In Chapter 4, we present work related to megamodel management, specifically, collection operators for megamodels.
- In Chapter 5, we detail our approach for heterogenous megamodel slicing constructed as a model-based workflow.

Part III. This part focuses on assurance case management. It is organized as follows:

- In Chapter 6, we introduce relevant background literature, concepts and notations related to standards and assurance case modeling notations and tools.
- In Chapter 7, we present our approach for assurance case reuse due to system evolution, which is a generic approach (i.e., not assurance case notation dependant, and not automotive specific). The approach is demonstrated on the power sliding door example for validation.
- In Chapter 8, we present our specialized approach for GSN and ISO 26262. We also present a set of improvement techniques that ensure a more precise impact assessment result.

Part IV. This part presents tool support and a case study, and is organized as follows:

- In Chapter 9, we present *MMINT-A*, a tool we have developed as an extension to *MMINT* which allows us to assess the impact of system design changes on assurance cases.
- In Chapter 10, we use *MMINT-A* on a bigger example from the automotive domain, namely, the Lane Management System (LMS) case study.

Part V concludes the thesis, where Chapter 11 summarizes the work presented and outlines directions for future research.

Various parts of this thesis have been previously peer-reviewed and published. Below, I outline these parts and my contributions to each of them:

- Chapter 1 includes content published in [Kokaly *et al.*(2016b)] and [Kokaly(2017)], both of which I was the main author of, and the ideas presented there were my own.

- Chapter 2, Chapter 3, Chapter 4 and Chapter 5 include content published in [Diskin *et al.*(2013), Salay *et al.*(2015), Salay *et al.*(2016), Di Sandro *et al.*(2015)], all of which were publications I contributed to as a co-author, however, they are presented here as they form foundations for much of the work in the thesis. In particular, in Chapter 3, I did not do any tool implementation myself, but was involved in specifying the requirements and supervising the implementation, testing and running the experiments needed for this thesis. In Chapter 4, my contributions were in defining the behaviour of the operators, applying them on all the scenarios presented, specifying their implementation and guiding with the experiments.

- Chapter 6 contains content published in [Maksimov *et al.*(2018)], which was a survey conducted by a masters student, where I contributed to the work by providing guidance and feedback on the assurance case tools in the literature and how to assess them and present the survey results in a useful manner.

- The content of Chapter 7 has been published in [Kokaly *et al.*(2016a)], which is one of the main contributions of this thesis.

- The content of Chapter 8 has been published in [Kokaly *et al.*(2017)], which is another main contribution of this thesis.

- The content of Chapter 9 has been published in [Fung *et al.*(2018)], where a masters student who did the majority of the tool development wrote the tool paper. I contributed to this work as a co-author, guiding the structure and content, including creation of the example presented in the paper. The specifications for the tool came from my own work and ideas, and the development was done by the student. I participated in the testing and evaluation of the tool.

- The LMS specification and models in Chapter 10 came from the literature [Blazy *et al.*(2014)], the safety case was constructed by me, and the change scenarios were specified by me as well. A masters student helped run the experiment using the *MMINT-A* tool, and the results and conclusions were written by me.

Part II

Megamodel Management

Chapter 2

Background: Model Management

In this chapter, we present a number of core concepts required for describing our contributions in the rest of this thesis. First, Section 2.1 presents the example of a Power Sliding Door (PSD) system, which will be used throughout the rest of the thesis. Then, Secton 2.2 presents background material on models, megamodels and model management. Section 2.3 talks about model slicing, and Section 2.4 presents some general material on model evolution, both of which are techniques we will use in the assurance case change impact assessment approach in later chapters.

This chapter includes content published in [Diskin *et al.*(2013), Salay *et al.*(2015), Salay *et al.*(2016)], all of which were papers I contributed to as a second author, however, they are presented here as they form foundations for much of the work in the thesis.

2.1 Running Example: Power Sliding Door System

We use the Power Sliding Door (PSD) system, presented in Part 10 of the ISO 26262 standard [ISO(2011)], as a running example thoughout this thesis. PSD is an automotive subsystem that controls the behaviour of a power sliding door in a car. The system has an *Actuator* that is triggered on demand by a *Driver Switch*. As per the standard, the power sliding door system is considered an *item*, with an architecture shown in Figure 2.1. The *Driver Switch* input is read by a dedicated Electronic Control Unit (ECU), referred to as *AC ECU*, which powers the *Actuator* through a dedicated power line. The vehicle equipped with the item is also fitted with an ECU which is able to provide the vehicle speed. This ECU is referred to as *VS ECU*. The system includes a safety element, namely, a *Redundant Switch*. Including this element ensures a higher level of integrity for the overall system.

As shown in Figure 2.1, the *VS ECU* provides the *AC ECU* with the vehicle speed. The *AC ECU* monitors the driver's requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so, commands the *Actuator*. The *Redundant Switch* is located on the power line between the *AC ECU* and the *Actuator*. It switches on if the speed is less than or equal to 15 km/h, and off whenever

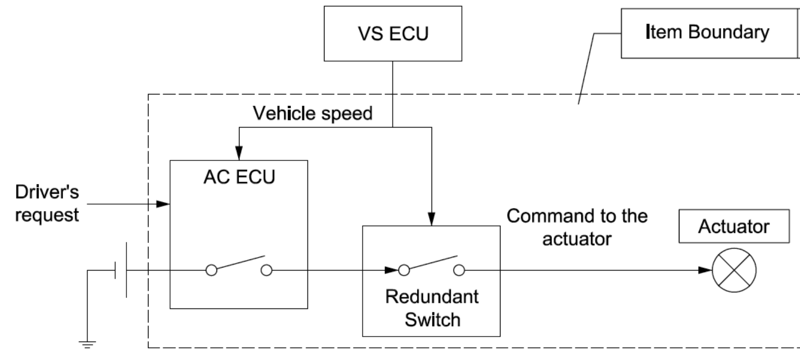


Figure 2.1: Power sliding door system with redundancy [ISO(2011)].

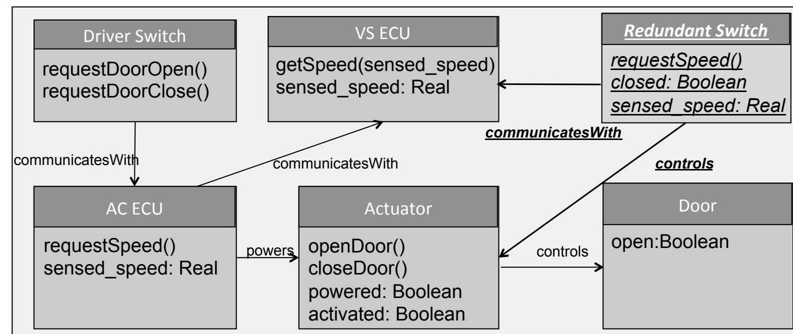


Figure 2.2: Power sliding door system class diagram.

the speed is greater than 15 km/h. It does this regardless of the state of the power line (its power supply is independent). The *Actuator* operates only when it is powered.

We present the system design as a combination of a class diagram (see Figure 2.2) that describes the various components, their attributes, methods and relationships between them, and a sequence diagram (see Figure 2.3) which describes the behaviour of the system. There are three threads running in parallel in the sequence diagram: the top thread describes the behaviour of the *Redundant Switch*; the middle thread describes the behaviour when the driver requests to open the door, and the bottom thread describes the behaviour when the driver requests to close the door. The traceability between the two models is given implicitly by the sequence diagram referencing objects which are instances of the classes in the class diagram.

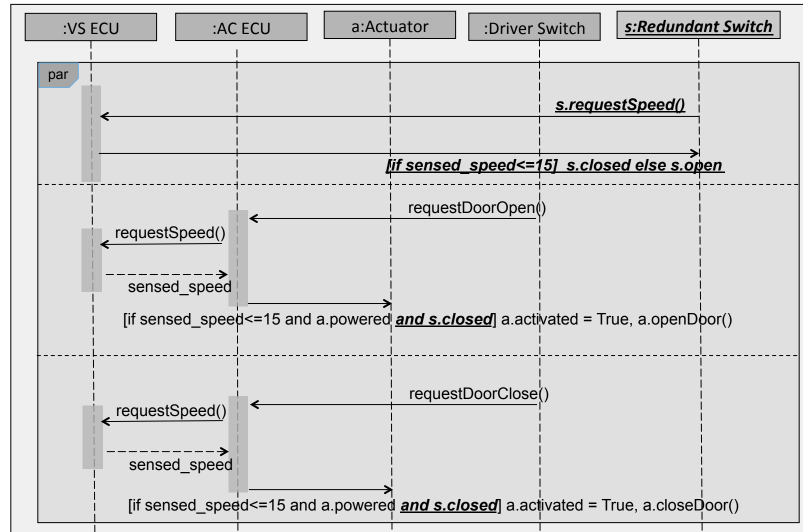


Figure 2.3: Power sliding door system sequence diagram.

2.2 Modeling and Model Management

This section contains some background on modeling as well as required definitions to be used in future chapters.

2.2.1 Modeling

Model-Driven Engineering (MDE) is an approach to software development in which software models play a primary role. MDE allows developers to work and reason about software requirements, design, and correctness at higher levels of abstraction, and to automatically generate implementations, deployments, and other artifacts. This section explains the basic terms and concepts that MDE is built upon, and that will be used throughout this thesis.

Model. The term *model* is derived from the Latin word modulus, which means measure, rule, pattern, example to be followed [Ludewig(2004)]. A model is therefore an abstract representation of a system's structure, function, or behaviour. A useful model serves as means of communication between team members, and as it is cheaper to build than the actual system, it aids in system analysis and testing. Two key attributes to effective modelling are *abstraction* and *classification*. Abstraction means ignoring information not relevant in a particular context, and classification means grouping important information based on common properties. MDE models are usually defined in UML, but could be defined in any other general purpose or domain specific modelling language. In software engineering, models appear in all areas and applications such as design models, process models, analysis models, documentation models. etc.

Metamodel. A *metamodel* is a more abstract model that defines the structure, semantics and constraints for a set of models. This set includes models that share common syntax and semantics. One can also think of a metamodel as the model of a modelling language, as it provides a way of describing the entire class of models that can be represented by that language. Thus, each model is considered an "instance of" some metamodel, and recursively, each metamodel is an instance of a model that describes it, called a *meta-metamodel.* In theory, one can recursively define infinite levels of metamodeling, but in practice, it has been shown that meta-metamodels can be defined based on themselves, and, therefore, are usually at the highest level of abstraction.

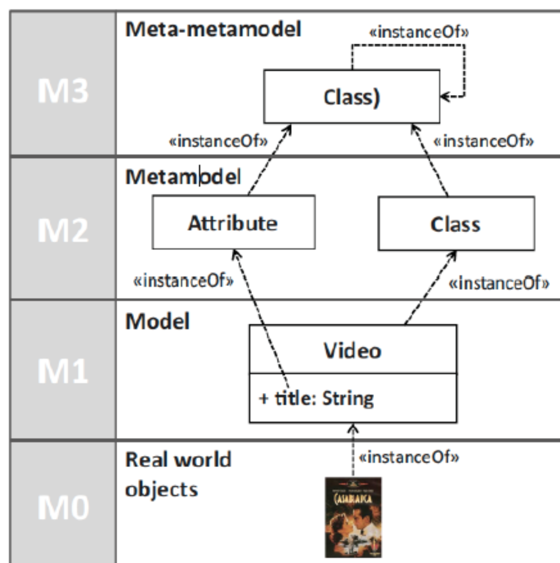


Figure 2.4: The Four-layer Metamodel Hierarchy. Source: [Brambilla *et al.*(2012)].

In the same way a computer program is said to "conform to" the grammar of the programming language it is written in, we say that a model "conforms to" its metamodel, etc. Figure 2.4 from [Brambilla *et al.*(2012)] shows an example of how a model (M1) of a real world object (M0) relates via "instance of" relations to its metamodel (M2) and meta-metamodel (M3). Note that at the highest level of abstraction is the "Class" object, which is an instance of itself. Metamodels are generally useful for defining new modeling languages and new properties or features to be associated with existing information (i.e., metadata). They are also important when integrating models and in other model management operations.

Platform. A *platform* is defined as the specification of an execution environment for a set of models. A platform usually has a realization (i.e., implementation) of the specification it represents. As an example, consider the UML metamodel which describes how UML models are structured, the elements they contain, and the properties those elements have. This metamodel can then describe properties of a particular platform, which could have multiple metamodels describing it. Figure 2.5

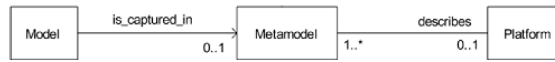


Figure 2.5: Models, metamodels, and platforms. Source: [Mellor *et al.*(2004)].

from [Mellor *et al.*(2004)] depicts this relationship. A well-defined application architecture, including its runtime system, can also be a platform for applications. This is often considered to be one of the key concepts for Model Driven Software Development as stated in [Stahl *et al.*(2006)].

Model Mapping. A *mapping* between models takes as input one or more "source" models and produces one "target" model as output. There are rules, called *mapping rules*, which constrain the mapping through a *mapping function*. The rules on model mappings are defined at the metamodel level and apply to all sets of source models that conform to the given metamodel. Figure 2.6 from [Mellor *et al.*(2004)] shows using an UML class diagram, how the various MDE objects defined so far are related.

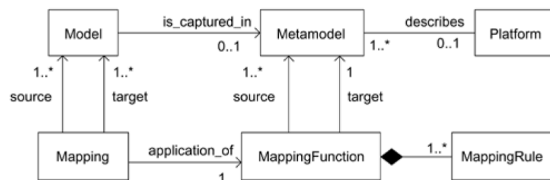


Figure 2.6: Models, mapping functions, and mapping rules. Source: [Mellor *et al.*(2004)].

Model Transformation. Once we have the source and target model definitions, and once a model mapping is also defined along with mapping rules, a model *transformation* can be generated using these rules. Model transformations can then themselves be viewed as models and managed as such, having their own metamodels. A model transformation language (the metamodel of the transformation) would then conform to the same meta-metamodel as the model it transforms [Brambilla *et al.*(2012)].

2.2.2 MDA, MOF, EMF and Ecore

The *Model Driven Architecture* (MDA) [Object Management Group(2014)] is a software development methodology introduced by the Object Management Group (OMG) in 2001. Relying on a series of OMG standards, such as the Meta Object Facility (MOF) [OMG(2015)], the Unified Modeling Language (UML) [Object Management Group(2015a)], the XML Metadata Interchange (XMI) [Object Management Group(2015b)], and others, the MDA is OMG's realization of Model Driven Engineering (MDE).

In MDA, models, metamodels and model transformations are first class development artifacts.

Software applications are created by applying transformations on models expressed at a high level of abstraction to derive models at lower levels of abstraction.

The MOF (MetaObject Facility) specification [OMG(2015)] states that a metamodel consists of *element* types, and that each element type has zero or more *reference* and *attribute* types. Then, given a model M of a metamodel T, an *atom* of M denotes any element, reference or attribute in M and $atoms_M$ denotes the set of all atoms in M. For example, in a UML class diagram, Class is an element, OwnedBy is a reference and IsAbstract is an attribute. For a more comprehensive description of the MOF specification, please refer to [OMG(2016)].

The Eclipse Modeling Framework (EMF) [Eclipse(2018a)] is a framework and code generation facility in Eclipse for building tools and other applications based on a structured data model. EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF is used for the implementation of *MMINT* described in Chapter 3, which we then extend into the tool *MMINT-A* in Chapter 9.

Ecore is the core (meta-)model at the heart of EMF. It allows expressing other models by leveraging its constructs. Ecore is also its own metamodel (i.e., Ecore is defined in terms of itself). Ecore is considered to be the defacto reference implementation of OMG's EMOF (Essential Meta-Object Facility). The Ecore metamodel is shown in Figure 2.7[1].

Note from Figure 2.7 that the Ecore metamodel contains the following constructs:

- EClass: represents a class, with zero or more attributes and zero or more references. We refer to its instances as *classes*.
- EAttribute: represents an attribute which has a name and a type. We refer to its instances as *attributes*.
- EReference: represents one end of an association between two classes. It has flags to indicate if it represents a containment and a reference class to which it points. We refer to its instances as *references*.
- EDataType: represents the type of an attribute, e.g., int, float or java.util.Date. We refer to its instances as *types*.

2.2.3 Model Management

A complexity problem in MDE arises due to the proliferation of software models. As such, the area of Model Management [Bernstein(2003)] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their *relationships* (i.e., mappings between models) can be manipulated using *operators* (i.e., specialized model transformations) to achieve useful outcomes.

[1]Source: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

As mentioned in Chapter 1, model management operators that have been studied include the following:

• The *slice* [Nejati *et al.*(2012)] operator accepts a model and a *slicing criterion* and extracts the subset of the model satisfying the criterion. Model slicing is a way to manage model complexity by focusing on a relevant subset of a model.

• The *match* [Bernstein(2003)] operator accepts two models and produces a relationship containing mappings between equivalent (or similar) elements in the models. This is usually interpreted as identifying the overlap between the models.

• The *diff* [Bernstein(2003)] operator accepts two models and produces a model that represents the differences between the models. Model differencing aids the comparison of model content, e.g., across different versions.

• The *merge* [Brunet *et al.*(2006)] operator accepts two models and a relationship expressing the overlap between them and produces a model that combines the content of the models according to the overlap. Model merge must address the issue of conflicts that could occur when the content is combined.

• The *lift* [Salay *et al.*(2014)] operator accepts a model transformation and produces a product line transformation that behaves the same way as the original model transformation for each product in the product line. Transformation lifting saves effort by allowing model transformation to be reused for product lines of models.

• The *filter* [Salay *et al.*(2015)] operator accepts a megamodel and a model (relationship) property and produces a megamodel with all models (relationships) not satisfying the property removed. Filtering a megamodel is useful for managing the complexity of large collections of related models.

• The *map* [Salay *et al.*(2015)] operator accepts a megamodel and a model transformation to produce the megamodel that results from applying the transformation to all applicable models and relationships in the input megamodel. Mapping a transformation over a megamodel is used to reduce the effort of applying a transformation to the elements of a large collection of related models.

Each of these operators can be viewed as an *abstract* transformation that defines a class of concrete transformations (i.e., the implementations that refine the operator for particular model types). For example, a model merge of class diagrams is implemented differently than a model merge of state machines. Another widely used class of transformations used in model management are *bidirectional transformations* [Diskin *et al.*(2010)]. Bidirectional transformations are used to keep two related models synchronized when one of the models changes (e.g., via model co-evolution, correction, etc.) by generating the update for the other model.

2.2.4 Definitions

Below are definitions of modeling concepts used in this thesis.

Definition 1 (Model) *A* model *is a set of typed elements and links conforming to a metamodel. We use the term* atom *to denote either an element or a link.*

Definition 2 (Model relationship) *A* model relationship *connecting models M and M' consists of a set of typed links conforming to a metamodel. Each link connects atoms of M to atoms of M'.*

Definition 3 (Traceability relationship) *A* traceability relationship *is a model relationship in which the links express a* dependency *relationship between the atoms it connects. The dependency relationship can be unidirectional or bidirectional depending on the type of traceability link.*

Note that the definition of traceability relationship used in this thesis is broader than what is typically used by requirements engineering [Gotel and Finkelstein(1995)] and narrower than what is sometimes used for general modeling [Aizenbud-Reshef *et al.*(2006)]. We focus solely on traceability relationships and use them to determine cross-model dependencies. In fact, we assume that the only relationships in the megamodels are the traceability relationships. In Section 5.2.3, we discuss how this assumption can be relaxed.

Definition 4 (Model fragment) *A* model fragment *S of a model M, denoted $S[M]$, is any subset of atoms of M.*

To help visualize and work with collections of models and their relationships, model management uses a special type of model called a *megamodel* [Diskin *et al.*(2013)] whose elements represent models and links between elements represent relationships between the models. Operators for megamodel management, namely, *filter*, *map*, and *reduce*, are presented in more detail in Chapter 4.

Figure 2.8 shows the simplified metamodel used for megamodels in this thesis. A Megamodel consists of a graph of named and typed Model elements with Relationship links connecting them. These refer to models and model relationships (defined above), respectively, and the types indicate their metamodels. The well-formedness constraint requires that the models on either end of every relationship are distinct. We have made the simplifying assumptions that (1) all relationships are binary; and (2) megamodels cannot be nested or reference other megamodels. In Section 5.2.3, we discuss how these assumptions can be relaxed.

Figure 2.9 shows a megamodel for our PSD example. Here, PowerSlidingDoor : CD and PowerSlidingDoor : SD refer to the Class Diagram and Sequence Diagram, respectively, while the line connecting them refers to the relationship of type CD − SD for connecting these two types of models. Note that relationship names are optional.

A modeling environment typically consists of various modeling artifacts stored in *a repository*. A *megamodel* then is a model whose elements refer to artifacts in the repository. Next, we formalize the concept of megamodel and give other necessary definitions.

Basic Types. We define the concept of "mega-graphs" – *mgraphs*. A megamodel is an mgraph whose nodes refer to artifacts in the repository.

Definition 5 (mgraph) *An* mgraph *is a structure that is an instance of the metamodel in Figure 2.10. Given an mgraph G, we write G_C to denote the set of nodes in node class C. When C*

is omitted, the node class is `Node`*. We use abbreviations* `Mod` *and* `Rel` *for node classes* `Model` *and* `Relationship`*, respectively. For $n \in G$, we write $n.R$ to denote the set of nodes on the other end of reference*[2] *R from node n.*

Note that our mgraph metamodel conforms to the Ecore metamodel shown in Figure 2.7, where boxes are instances of EClasses, arrows are instances of EReferences, and "type" and "name" are instances of EAttributes.

In this thesis, we limit our focus to megamodels that can refer to artifacts corresponding to the concrete node classes in Figure 2.10. A *relationship* is a mapping between two or more models on its ends. A *transformation application* is the record of having performed a given transformation on a set of input models and relationships to produce a set of output models and relationships. We make no further assumptions about the way models, relationships or transformation applications are represented or what they contain. The "mega" versions of these artifacts: megamodels, megarels and megaApps are defined below. We assume the existence of a repository.

Definition 6 (Repository) *A* repository $\mathscr{R}$ *is a store for artifacts that is itself structured as an mgraph of artifacts (i.e., rather than an mgraph of symbols). Thus, a relationship has references to the models on its ends, etc.*

Models, relationships and transformation applications are typed by model types, relationship types and transformations, respectively. We assume that type compatibility (e.g., via sub-typing) is given by a relation `TypeComp`.

Definition 7 (Type compatibility) *Given a type compatibility relation* `TypeComp`*,* $\texttt{TypeComp}(T, T')$ *indicates that an artifact of type T can be used wherever the type T' is required. The relation* `TypeComp` *must be reflexive.*

Mappings between mgraphs are called mgraph homomorphisms.

Definition 8 (mgraph homomorphism) *Given mgraphs G, G' and type compatibility relation* `TypeComp`*, an* mgraph homomorphism *$f : G \to G'$ is a function $f_{\texttt{Node}} : G_{\texttt{Node}} \to G'_{\texttt{Node}}$ that satisfies the following conditions for preserving all node classes C, references R and types:*

1. $\forall n \in G \cdot n \in G_C \Rightarrow f_{\texttt{Node}}(n) \in G'_C$
2. $\forall n, n' \in G \cdot n' \in n.R \Rightarrow f_{\texttt{Node}}(n') \in f_{\texttt{Node}}(n).R$
3. $\forall n \in G_{\texttt{TypedNode}} \cdot \texttt{TypeComp}(n.\texttt{type}, f_{\texttt{Node}}(n).\texttt{type})$

A typed mgraph homomorphism *is one where* `TypeComp` *is equality. An* mgraph isomorphism *is one where $f_{\texttt{Node}}$ is a bijection.*

[2] Please refer to description of Ecore references in Section 2.2.2.

Condition (1) ensures that f preserves node classes and condition (2) ensures that f preserves the endpoints of references. These are standard conditions for a homomorphism to be a structure-preserving mapping. Condition (3) additionally ensures that for typed nodes, f preserves type compatibility of nodes. Note that node names need not be preserved by f.

Mega Types. Intuitively, all the mega types represent collections of artifacts.

Definition 9 (Megamodels) *Let a model repository $\mathscr{R}$ of artifacts be given. A* megamodel *is a pair $\langle G, d\rangle$, where G is an mgraph and $d : G \to \mathscr{R}$ is a typed mgraph homomorphism, called the* dereferencing mapping*, that maps the nodes of G to the artifacts they represent in $\mathscr{R}$.*

When it is clear from the context, we will use a megamodel interchangeably with its mgraph.

Definition 10 (Megarel) *A* megarel *is a megamodel restricted to containing only* `Relationship` *and* `Megarel` *nodes (refer to Figure 2.8), but* `end` *references of these nodes are contained within the ends of the megarel.*

Thus, a megarel is the "mega" version of a *relationship*, and can be viewed as a"relationship-like" collection that itself has megamodels on its ends.

Definition 11 (MegaApp) *A* megaApp *is a megamodel restricted to containing only* `Transformation Application` *and* `Transformation MegaApp` *nodes (refer to Figure 2.8), but the* `in` *and* `out` *references of these elements are contained within the input and output connections of the megaApp. That is, both megarel and megaApp artifacts are connected to other artifacts in $\mathscr{R}$.*

Thus, a megaApp is the "mega" version of a *transformation application*, and can be viewed as a "transformation-application-like" collection that represents a record of having performed a given transformation on a set of input megamodels and megarels to produce a set of output megamodels and megarels.

Figure 2.11 gives an example of a repository including three megamodels, a megarel (as well as other artifacts shown as shaded boxes). To avoid visual clutter in this example, the dereferencing mappings are not shown but are implied by the names; however, in general, names across the mapping may be different. The examples of the megamodels and megarels show the concrete syntax we use for illustrations in this chapter. Models are shown as boxes, relationships are shown as diamonds with binary relationships shown optionally as a line. A transformation application is given as an oval, with the input elements connected with arrows pointing into the oval and output elements connected with arrows pointing out of the oval. All models, relationships and transformation applications have a label of form *name:type*, where the name is optional. Megamodels, megarels and megaApps are shown similarly as their non-"mega" counterparts but with **`thick`** borders. Furthermore, these elements are not typed. For example, in megamodel `X` at the top of the figure, the box with label `C : CD` refers to the class diagram with name `C`. The diamond `R2 : CDrel` refers to the corresponding `CDRrel` artifact with name `R2`, the thick bordered box labeled `X1` refers to the megamodel `X1` shown

below it which itself refers to models `B` and `E`, etc. No megaApp is shown here, but this will be illustrated in Chapter 4.

Properties and Transformations. Models can satisfy properties and participate in transformations. We define these below.

Definition 12 (Property) *A* property *is a constraint on an artifact. Given an artifact A and a property P, we write $A \models P$ to denote that A satisfies P. Every property is defined for an artifact of a specific type. If A has type T, P has type T' and the type compatibility relation* `TypeComp` *is given, the following condition must hold:* $(A \models P) \Rightarrow \texttt{TypeComp}(T, T')$.

Thus, we assume that artifacts not compatible with the type of the property do not satisfy the property.

Definition 13 (Transformations) *A* transformation *is a function that maps an mgraph of models and relationships to another mgraph of models and relationships. Given a transformation F, the* signature *of F is a pair $\langle I, O\rangle$ where $I \cup O$ is an mgraph, I is an mgraph called the* input signature *and O is a set of mgraph nodes called the* output signature*.*

Note that we make no assumptions about what language is used for expressing properties or defining transformations. Also note that O is a set of mgraph nodes and not an mgraph, as it contains not only the mgraph output, but also the relationships back to the input mgraph (i.e., O is not a well-formed mgraph). To better explain this, Figure 2.12 gives an example of the signature for a transformation `CDMerge` that accepts two class diagrams and a relationship between them and produces the merged class diagram with relationships back to the original two class diagrams. The input signature consists of the models a, b and relationship r and the output signature has model ab with relationships ra and rb. Written textually, the signature consists of $I = \{\texttt{a} : \texttt{CD}, \texttt{b} : \texttt{CD}, \texttt{r}(\texttt{a}, \texttt{b}) : \texttt{CDrel}\}$, and $O = \{\texttt{ab} : \texttt{CD}, \texttt{ra}(\texttt{a}, \texttt{ab}) : \texttt{CDrel}, \texttt{rb}(\texttt{b}, \texttt{ab}) : \texttt{CDrel}\}$. Even though a relationship is an output of a transformation, it can connect input elements. For example, both output relationships ra and rb are connected to inputs.

Definition 14 (Transformation binding) *Given transformation F with signature $\langle I, O\rangle$, a* binding *K of F is a megamodel $\langle I, d_I\rangle$ where d_I is an mgraph homomorphism (rather than a* typed *mgraph homomorphism). We write $F(K)$ to denote the corresponding megamodel $\langle I \cup O, d_{I\cup O}\rangle$ giving the result of applying F to K. We say that F is* commutative *if for every pair of isomorphic bindings K, K' (i.e., mgraph isomorphisms of I), $F(K)$ is isomorphic to $F(K')$.*

Thus, a binding K of F assigns artifacts to the nodes of its input signature I and then $F(K)$ can be evaluated to assign newly created artifacts to the nodes of the output signature. `CDMerge` in Figure 2.12 is an example of a commutative transformation – given any two class diagrams `M1`, `M2` related by a relationship `R`, the merged output is the same regardless of whether we use the binding $\{\texttt{a} := \texttt{M1}, \texttt{b} := \texttt{M2}, \texttt{r} := \texttt{R}\}$ or $\{\texttt{a} := \texttt{M2}, \texttt{b} := \texttt{M1}, \texttt{r} := \texttt{R}\}$.

Definition 15 (binding with megamodel) *Given megamodel* $X = \langle G_X, d_X \rangle$ *and transformation* F *with signature* $\langle I, O \rangle$*, a* binding of F within X *is the megamodel* $\langle I, d_X \circ b \rangle$ *where* b *is an injective mgraph homomorphism* $b : I \rightarrow X$*.*

That is, a binding of F within X is formed by finding a set of nodes in X that match I.

Finally, a megamodel fragment is defined as containing only model fragments and no relationships between them. We define it formally as follows:

Definition 16 (Megamodel fragment) *A* megamodel fragment S *of a megamodel* X*, denoted* $S[X]$*, is any set of model fragments of the models in* X*. We say that* $S[X]$ *is contained in* $S'[X]$*, denoted* $S[X] \sqsubseteq S'[X]$*, iff the following condition holds:*

$$\forall M \in X \cdot \cup\{S[M] | S[M] \in S[X]\} \subseteq \cup\{S'[M] | S'[M] \in S'[X]\}$$

Thus, $S[X] \sqsubseteq S'[X]$ when, for each model $M \in X$, the combined model fragments of M in $S[X]$ is contained in the combined model fragments of M in $S'[X]$.

2.3 Model Slicing

Program slicing, and, correspondingly, model slicing approaches, fall into four categories: *static*, *dynamic*, *conditional* and *amorphous* [Clark(2011)]. In each case, we are given a model and a slicing criterion indicating some "aspect of interest" in the model, and the slicing process produces a slice of the model that addresses the criterion. Static slicing uses a model fragment as the criterion. A *forward slice* expands the criterion to all dependent atoms while a *backward slice* expands to all depending atoms. While static slicing uses a subset of the syntax as a criterion, dynamic slicing uses a constraint from the semantic domain. For example, dynamic slicing can be used to identify the classes used in a particular run of a program. Conditional slicing combines both static and dynamic approaches by allowing a hybrid criterion. Finally, while the first three types of slicing produce a slice that is a fragment of the model, amorphous slicing allows the slice to be a different model. For example, the approach used in [Nejati *et al.*(2012)] adds stuttering transitions to state machine slices in order to preserve behaviours.

In this thesis, we focus on static forward slicing since it is readily applicable to assessing the impact of changes due to model evolution. We define this as follows.

Definition 17 (Static forward model slice) *Given a model* M *and model fragment* $S[M]$*, the* static forward slice *of* M *with respect to the slicing criterion* $S[M]$ *is the model fragment* $S'[M]$ *satisfying the following conditions:*

1. *(Correctness)* $S'[M]$ *contains all atoms of* M *that are directly or indirectly dependent on atoms of* $S[M]$*.*

2. *(Minimality) Every atom of* $S'[M]$ *is either directly or indirectly dependent on atoms of* $S[M]$*.*

Note that since $S[M]$ is dependent on itself, these conditions imply that $S[M] \subseteq S'[M]$.

2.4 Model Evolution

In MDE, model evolution is studied in order to understand why models change and how that impacts consistency of related models. Examples of kinds of model evolution changes are presented in a survey on the evolution of UML models [Khalil and Dingel(2013)]. In this thesis, we are interested in three of the presented types: *evolution due to fixing errors*, *evolution due to changing functionality*, and *evolution due to changing model quality*.

The general approach we use is to determine what parts of the assurance case are impacted by the change. Then the new assurancce case must redo these parts and potentially can reuse the unimpacted parts. Depending on the type of change we are considering, the impact assessment is different. In the case of fixing errors, this means that the current assurance case was either incorrect or incomplete (or both) because it did not catch the error. This points to the need to address two questions: (1) why the assurance case was not adequate, and (2) how to change the assurance case to address this type of change. For the former, this requires an analysis of the process followed to produce the original assurance case and a decision on its causes. We consider this to be outside of the scope of our work. For the latter, this requires an assessment of the impact of the change in the system and then the corresponding impact in the assurance case. The impact of the incomplete/incorrect parts of the assurance case also need to be determined if these are different than the impact of the error fix. In the case of changing functionality, this means that the requirements must have also changed. Thus we must do an impact analysis of both the changed requirements and of the system changes and how these correspondingly impact the assurance case. Finally, in the case of changing model quality, this means that the existing system was adequate, so the assurance case was not flawed and the requirements have not changed. In this case, we just assess the impact of the change in the system and the corresponding impact on the assurance case.

2.5 Chapter Summary

In this chapter, we have presented a running example (PSD) which will be used to demonstrate our ideas in subsequent chapters. We have also presented core concepts on modeling, megamodels, and model management, with a focus on model slicing and model evolution. These concepts form the basis for much of the material presented in the rest of this thesis.

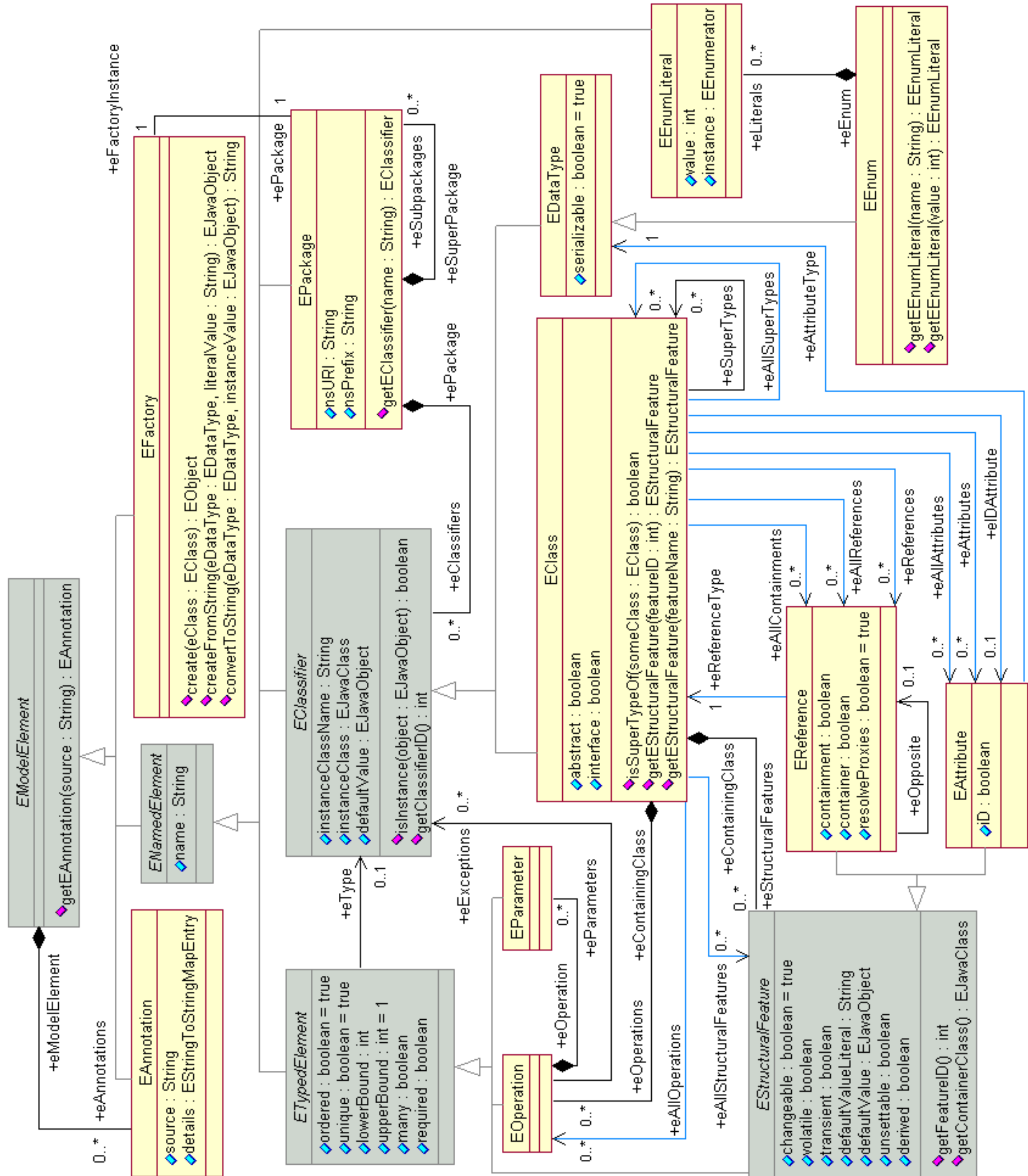


Figure 2.7: Ecore components and their relations.

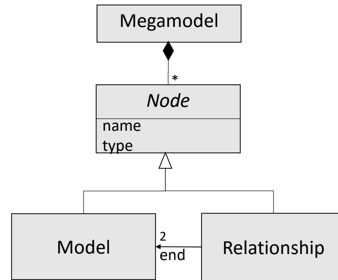


Figure 2.8: (Simplified) metamodel for megamodels.

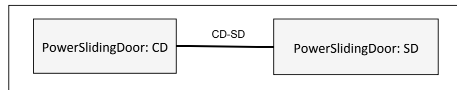


Figure 2.9: Power sliding door system megamodel.

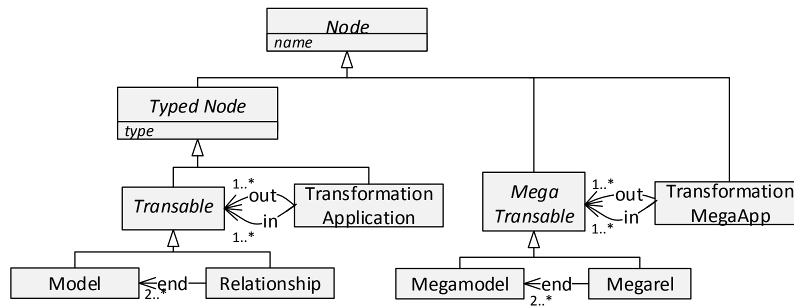


Figure 2.10: Metamodel of an mgraph.

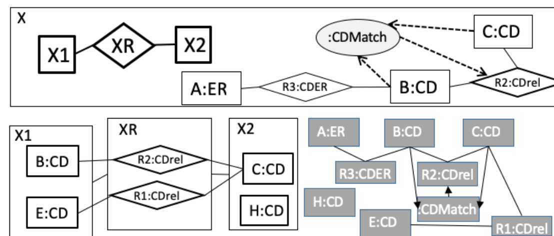


Figure 2.11: An example of a repository including megamodels and a megarel showing the concrete syntax.

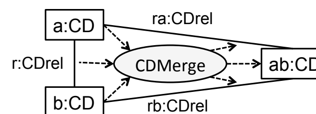


Figure 2.12: Signature of a transformation `CDMerge` for merging class diagrams.

Chapter 3

Background: *MMINT*

Modern software development requires managing multiple, heterogeneous software artifacts, but the proliferation of models creates an accidental complexity that must be managed. Model Management [Bernstein(2003)] addresses this challenge. This is realized through the use of special models called *megamodels* [Bézivin *et al.*(2004)] to represent model management scenarios at a high level of abstraction and general purpose transformations (also called *operators*) [Brunet *et al.*(2006)] to manipulate models and their relationships (i.e., mappings between models) in order to perform useful tasks. For example, a model *match* operator finds correspondences between two models and packages these as a mapping between the models. A *merge* operator can be used to combine the content of the two models using the correspondence information in the mapping.

Several model management frameworks have been developed to support Model Driven Engineering (MDE) by facilitating the development of operators and the implementation of specific model management tasks. For example, Epsilon [Kolovos *et al.*(2015)] provides multiple domain-specific languages, each specialized for a different model management task such as merge, validation, transformation, etc. The Atlas suite of tools [Bézivin *et al.*(2005b)] is centred around the ATL transformation language and its use in different model management tasks. These frameworks focus on the *programming required to prepare for model management* rather than the *user environment required to carry out model management tasks*. However, such an environment is a key factor for practical model management.

An effective user environment requires a rich *interactive user interface* and *automated user assistance*. We elaborate on these in the context of model management:

- *Interactive User Interface* - Megamodels provide the means for raising the level of abstraction to manage complexity in model management, thus they are the natural user interface for model management. Furthermore, the user should be able to interact with the *type level* where model types (i.e., metamodels), relationship types and transformations are defined as well at the *instance level* where particular model management scenarios are expressed and carried out. *Interactivity* is an important characteristic of the user interface. Due to the

complexity of model management tasks, they often can only be partially automated. For example, performing a match between models may need to be manually verified to ensure the correspondences are sensible, and performing a merge of models may yield conflicts that need to be manually resolved using human judgment.

- *Automated User Assistance* - Many model management activities are one-off tasks that suggest the need for low effort rapid implementation, possibly with manual steps, rather than a polished and fully automated process. Even activities that become fully automated often initially require experimentation and exploration to get them right. User assistance to help adapt transformations for reuse, support reasoning about models, etc. can reduce effort and help manage complexity.

In this chapter, we describe the tool called *MMINT* (*"Model Management INTeractive"*) for graphical, interactive model management. *MMINT* is an existing tool which we use for running expirements to evaluate the operators in Chapter 4, and building an extension, *MMINT-A*, described in Chapter 9 for assurance case impact assessment. Next, we summarize the features of *MMINT* which address the requirements discussed above.

- *Interactive User Interface*
  - *instance level megamodel* - *MMINT* provides a customizable environment in which modellers can graphically create model management scenarios using megamodels at the instance level, interactively apply transformations and immediately see their result and, when necessary, manually drill-down into the detailed content of particular models or relationships.
  - *type level megamodel* - a megamodel is used at the type-level to visualize and allow rapid modifications of the type hierarchy of model types, relationship types and transformations at run-time to immediately impact instance level megamodels.
- *Automated User Assistance*
  - *megamodel operators* - the generic operators *map*, *filter* and *reduce*, available in many programming languages to simplify complex collection-based manipulation tasks, are built-in and adapted for use with megamodels as we will demonstrate in Chapter 4.
  - *type coercion* - a type coercion mechanism is available that automatically performs the necessary conversion transformations required to reuse transformations from related types.
  - *retyping* - a type down-casting mechanism is available that automatically detects cases when a model satisfies the constraints of a more specialized type and thus can use its transformations.

Organization. The rest of this chapter is organized as follows: In Section 3.1 we introduce and illustrate the features of *MMINT* by implementing a detailed model management scenario. In Section 3.2, the architecture of *MMINT* is described. Finally, in Section 3.3, we conclude with a summary of the chapter.

3.1 Using *MMINT* for Model Management

In this section, we use a model management scenario to illustrate the *MMINT* features addressing the requirements for a model management user environment discussed in the chapter introduction.

Illustrative scenario. Consider the following model management scenario. A company uses a megamodel to track its modeling artifacts (models and relationships between them). The company has determined that having public attributes in class diagrams is undesirable and now (1) would like to identify all class diagrams that contain this construct, (2) refactor them using a predefined transformation to remove the construct, (3) merge the modified class diagrams with the originals into a single class diagram, and finally, (4) produce a textual representation of the merged class diagram.

We now show how *MMINT* can be used to accomplish this scenario. Table 3.1 summarizes the *MMINT* features and the step of the scenario in which they are illustrated.

Executing the scenario. In *MMINT* a megamodel is referred to as a *MID* (Model Interconnection Diagram). *MMINT* uses a distinguished *Type MID* in which model types, relationship types and transformations are registered. Figure 3.1 shows a screenshot of the Type MID used to implement the scenario. Here, boxes represent model types and thick blue links between them are binary relationship types. The sub-typing between types is shown with the hollow-headed arrows. Transformations are ovals connected to their input and output types with named links (not shown to avoid clutter). For example, our scenario needs class diagrams and so ClassDiagram is a model type that is a sub-type of the general model type Model. The binary relationship type CDRel is a used to create mappings between ClassDiagram models. CDMatch is a transformation that takes two ClassDiagrams as input and produces a CDRel between them as output that contains links between

Requirement	***MMINT***	**Step**
Interactive User Interface	Instance megamodel	All
	Type megamodel	1
Automated User Assistance	Megamodel operators	
	map	2,3
	filter	1
	reduce	3
	Type coercion	4
	Retyping	1

Table 3.1: *MMINT* features and where they are illustrated in the scenario.

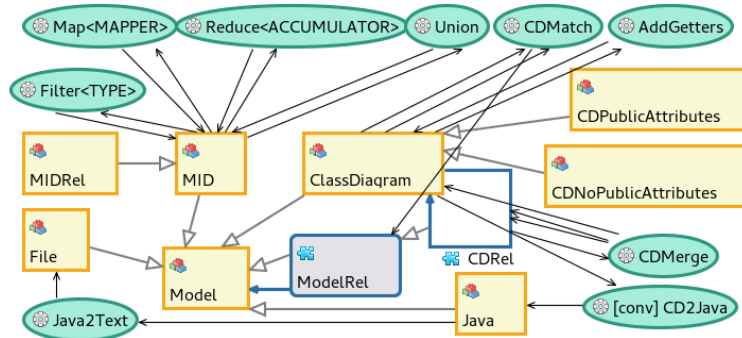


Figure 3.1: Type megamodel in *MMINT* used for the examples in this chapter.

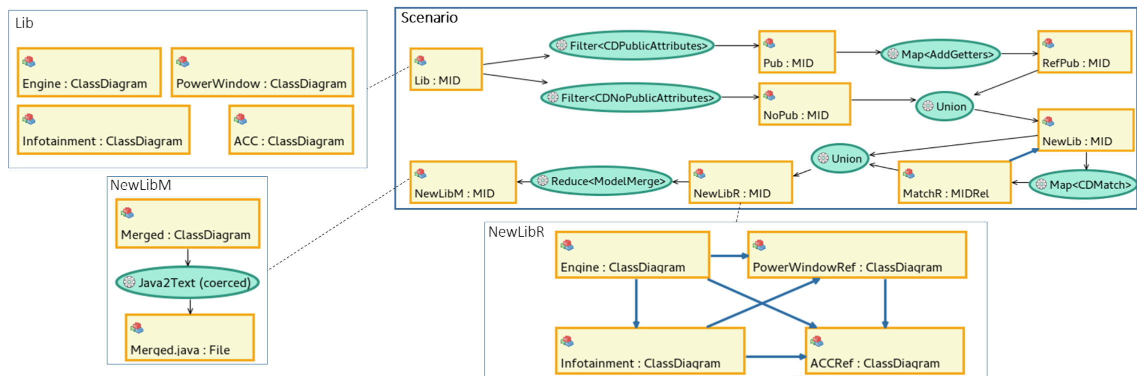


Figure 3.2: Screenshot of the final state of MID Scenario and selected other MID's used or created in the scenario.

classes that have the same name. Note that even the built-in megamodel operators Filter, Map and Reduce appear as transformations in the Type MID.

At the instance level, *MMINT* is used to create MIDs using a *MID editor* that allows an engineer to interactively create or import models, relationships and other MIDs, invoke transformations on them and inspect or change the results. We will incrementally and interactively build a MID called Scenario as we carry out the steps of the scenario. Figure 3.2 (top-right) shows a screenshot of the final state of this MID after step (4). Initially, Scenario contains only the top left-most box referring to MID Lib (top-left of Figure 3.2) that holds the class diagrams the company wants to process. We have limited the the number of CD's in MID Lib to four for this illustration; however, the scenario scales well (see [Salay *et al.*(2015)]).

Step (1). In the first step we need to separate out the class diagrams in Lib that contain public attributes. We can use the megamodel operator Filter ⟨TYPE⟩ that, when applied to a MID, removes all models/relationships that do not conform to type TYPE (see [Salay *et al.*(2015)] for details);

however, we need a model type that represents class diagrams that contain public attributes. The Type MID allows an engineer to create new types and use them immediately. Thus, we use the Type MID editor to define a new sub-type CDPublicAttributes of ClassDiagram containing the following OCL well-formedness constraint:

```
CDPublicAttributes:
  Attribute.allInstances()->exists(
    attribute | attribute.public)
```

Similarly, we create sub-type CDNoPublicAttributes with the negation of this constraint. Both are shown in Figure 3.1 as sub-types of ClassDiagram. We then split Lib into new MIDs Pub (containing CD PowerWindow and ACC) and NoPub (containing CD Engine and Infotainment) by applying Filter ⟨CDPublicAttributes⟩ and Filter ⟨CDNoPublicAttributes⟩, respectively, to Lib.

This step also illustrates the *retyping* feature of *MMINT*. Even though the class diagrams in Lib are typed as ClassDiagram, *MMINT* can check them against a sub-type (e.g., CDPublicAttributes) and automatically retype them if they conform. This feature can be invoked manually on any model by the engineer or internally by other operators – in this case, Filter. When invoked manually on a model (via right-clicking and a context menu), *MMINT* traverses the type hierarchy to check the model's conformance with all sub-types and returns the list of retyping options for selection by the user.

Step (2). In this step, we use a standard class diagram refactoring transformation, called AddGetters, that replaces each public attribute with a private attribute and a corresponding getter method. *MMINT* allows transformations in any language to be added to the Type MID. In this case, we have used Henshin [Arendt *et al.*(2010)] to implement AddGetters. We want to apply AddGetters to every class diagram in MID Pub. We can achieve this by using the megamodel operator Map ⟨MAPPER⟩ that applies a transformation given by parameter MAPPER to each model/relationship in a MID (see [Salay *et al.*(2015)] for details). In this case, we apply Map ⟨AddGetters⟩ to Pub to produce MID RefPub containing the refactored class diagrams. We then use the built-in operator Union to produce MID NewLib that combines the MID RefPub with the class diagrams containing no public attributes in MID NoPub. Thus, NewLib is the same as the original MID Lib but with refactored class diagrams.

Step (3). Now we wish to merge the class diagrams in NewLib into a single class diagram. We have available to us a class diagram merge transformation CDMerge that takes two class diagrams and a CDRel relationship between them indicating how the class diagrams overlap [Brunet *et al.*(2006)]. That is, if the relationship has a link between two classes then CDMerge will combine these classes into a single class in the output merged model. Unfortunately, our MID NewLib contains only class diagrams and no relationships indicating their overlap. We can fix this by first using the transformation CDMatch that creates a CDRel between two class diagrams and puts links between

classes with the same name. We could apply CDMatch manually to each pair of class diagrams in NewLib but to save time, we apply Map ⟨CDMatch⟩ to NewLib and get all the relationships in one step in the MIDRel MatchR. A *MIDRel* is a megamodel containing only relationships. Finally, we union these relationships with the class diagrams to produce MID NewLibR (content shown in Figure 3.2).

At any point in our scenario we can drill-down into models or relationships to examine results or manually intervene in the process. For example, we may want to manually check or modify individually relationships that Map ⟨CDMatch⟩ produced to confirm that the correct classes are linked. To do this, we would double-click on a relationship (thick blue arrow in NewLibR) and view or edit it using the *MMINT* relationship editor.

Now we can merge the models in NewLibR pairwise using CDMerge. To do this quickly we use the megamodel operator Reduce ⟨ACCUMULATOR⟩ that accepts an arbitrary "merging" transformation and keeps applying it until it can be applied no more (see [Salay *et al.*(2015)] for details). In this case we apply Reduce ⟨CDMerge⟩ to MID NewLibR to produce the MID NewLibM containing a single class diagram with the merged content of the class diagrams in NewLibR.

Step (4). In the final step of our scenario we wish to create a Java representation of the the merged class diagram in MID NewLibM. Assume we already have a transformation CD2Java that converts a class diagram to a Java model (i.e., based on a Java metamodel). In addition, we have a transformation Java2Text that produces the textual code of a Java model. One way to get the result we want is to apply these transformations in sequence on the merged class diagram in MID NewLibM. However, this step can be simplified by using the *MMINT* feature of *type coercion* that enables transformation reuse.

MMINT allows some transformations in the Type MID to be explicitly marked as *conversion* transformations and this is the case with CD2Java (see Figure 3.1). Normally, when a model is selected in a MID and right-clicked, *MMINT* computes the list of applicable transformations based on the Type MID and presents a list to the engineer to choose from. The type coercion features means that *MMINT* not only lists the transformations that directly apply to the model based on its type, but also transformations that could indirectly apply if the model was first converted using conversion transformations and it does the conversions automatically. In our scenario, this means that when we right-click on the merged class diagram in MID NewLibM, the transformation Java2Text will be available and if selected, *MMINT* runs the necessary `CD2Java` conversion behind the scenes and then deletes the intermediate result.

3.2 *MMINT* Architecture

MMINT, which is available at: `http://github.com/adisandro/MMINT`, is an evolution of the MMTF model management framework [Salay *et al.*(2007)] adding the Type MID, support for retyping, type coercion and megamodel operators. *MMINT* is implemented in Java and uses the Eclipse

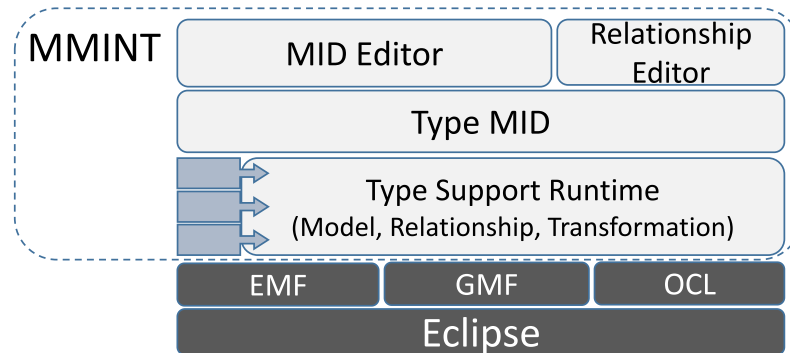


Figure 3.3: Architecture of *MMINT*.

Modeling Framework (EMF) [Steinberg *et al.*(2008)] to express models and the Eclipse Graphical Modeling Framework (GMF) to create custom editors for editing models and relationships. The overall architecture of *MMINT* is illustrated in Figure 3.3.

Model and Relationship types. Metamodels for new types can be plugged in or created directly through the Type MID. Implementations for supporting tools such as type-specific editors, validation checkers and solvers can also be plugged in and are managed by the type support runtime layer. A generic relationship editor is built into *MMINT*.

Transformations. New transformations can be implemented in Java or any transformation language and plugged into the type support runtime layer as transformation definitions. The definition includes metadata specifying the input and output types and other attributes such as if this is a conversion transformation used by the coercive typing feature. Transformations can also be designated as higher order and take types or transformations as parameters. The megamodel operators Filter, Map and Reduce are implemented in this way. Transformation metadata is stored in the Type MID (see Figure 3.1) for use at run-time.

MID Editor. The MID Editor provides the user interface through which MIDs are created and manipulated. Double-clicking on an element (model, relationship or MID) opens the corresponding artifact using the appropriate editor. Right-clicking on an element brings up a context menu that provides functionality appropriate to the corresponding artifact including retyping assistance, transformations that can be applied (including indirect ones via coercion) and validation.

3.3 Chapter Summary

In this chapter, we presented the interactive model management tool, *MMINT*, which allows users to perform automatically assisted model management tasks through the use of interactive type and instance megamodels. We illustrated its usage on a detailed model management scenario. *MMINT*

is used to evaluate our megamodel management operators presented in Chapters 4 and 5, as well as a basis for the assurance case impact assessment tool, *MMINT-A*, described in Chapter 9.

Chapter 4

Megamodel Management with Collection-Based Operators

In this chapter, we define operators on top of megamodels, which will be used as a basis for the assurance case impact assessment approach described later in the thesis.

This chaper contains material published in [Salay *et al.*(2015)], where I contributed to defining the behaviour of the operators, applying them on all the scenarios presented, specifying their implementation and designing the experiments.

Introduction

Model management has been studied from many perspectives including algebraic properties of operators [Brunet *et al.*(2006), Melnik *et al.*(2003)], categorical foundations [Diskin *et al.*(2013)], type theory [Vignaga *et al.*(2013)], megamodeling languages [Salay *et al.*(2009), Favre *et al.*(2012)] and practical implementations [Kling *et al.*(2012), Salay *et al.*(2007), Melnik *et al.*(2003), Kolovos *et al.*(2015)]. In these investigations, the focus is on the general manipulation of models rather than specifically on the manipulation of megamodels – since these are a kind of model, general model operators apply to them equally well. Yet other operators are needed due to the special role of megamodels in model management. Specifically, megamodels function as *collections* (of models and relationships) and so their manipulation should be like that of other collection types (e.g., lists, graphs, etc.) commonly found in modern programming languages. In particular, three collection operators are widely used: *map* for applying a function to every element of a collection, *reduce* for aggregating elements in a collection and *filter* for extracting a subset of the collection using a property as a selector. These operators would have great utility if available in the model management context. But while megamodels bear similarity to collections in programming, they also have their unique challenges that limit our ability to apply these techniques

without some adaptation. We illustrate these below.

A Motivating Scenario. A company uses a megamodel to track its modeling artifacts (models and relationships between them). The company identified a particular construct of some of its models as undesirable (e.g., multiple inheritance in class diagrams), and now (1) would like to identify all models that are of type class diagram and contain this construct, (2) refactor them using a predefined transformation to remove the construct, and (3) merge the modified class diagrams in order to compare the result to the merged version of the original bad class diagrams. A natural way to execute these steps is to (1) use *filter* to extract the bad class diagrams *and the relationships between these*, (2) use *map* to apply the refactoring transformation to these and *also allow the use of the corresponding refactoring transformation for the relationships* and, (3) use *reduce* with a merge transformation to combine all the refactored models pairwise, *correctly taking into account the relationships between them.* The *reduce* with merge also can be used to combine the original models. Thus, we need collection operators to manipulate the entire *graphs* of related models rather than just lists of models. Furthermore, we need to allow invoking *map* and *reduce* with transformations that can accept graphs of models and relationships as input and produce these as output.

Contributions. This chapter makes the following contributions:

1. We define the set of megamodel collection operators which treat relationships between models as first class entities:
   - **map** – for applying a transformation to the elements of a megamodel;
   - **reduce** – for aggregating the elements of a megamodel using a transformation; and
   - **filter** – for extracting a subset of elements of a megamodel that satisfy a property.
2. We analyze the complexity of the operators and discuss their scalability.
3. We demonstrate the approach by using them to express several non-trivial megamodel management scenarios.
4. We report on an implementation of the operators.

Organization. The rest of this chapter is organized as follows: After fixing the terminology in Section 4.1, we define the three collection operators for megamodels in Section 4.2. Section 4.3 illustrates these on four practical scenarios. Section 4.4 analyzes the complexity of the operators. Section 4.5 describes tool support. We compare our approach with related work in Section 4.6 and conclude in Section 4.7 with a summary of the chapter.

4.1 Traditional Megamodeling Operators

A number of model management operators have been defined, with *match*, *merge*, *diff*, and *slice* among them. For the illustrations in this chapter, we require only one of them – a simple type of megamodel merge that we call **union**. We define it in more detail below. The **union** operator combines the content of a set of megamodels into a single megamodel in which elements that refer to the same artifact are merged into a single element.

There are two possibilities for the set of input megamodels: (1) either they are an mgraph of megamodels and megarels, or (2) they are a set of megarels that share the same endpoints. Figure 4.1 illustrates both cases. In case (1), the result is a megamodel while in case (2) it is a megarel with the same endpoints as the inputs.

The union process can cause conflicts coming from the following two sources. If megamodel elements refer to the same artifact but the names of these elements differ, it is not clear which name to use for the merged element. To resolve this, we assume that the names in the union is a combination of the original names. Another conflict occurs when different artifacts are referred to by different elements using the same name. In this case, we assume that the names are made distinct in the union. Both of these conflict scenarios are illustrated at the bottom of case (1) in Figure 4.1. Both `A` and `D` refer to the same model and in the union the name `A_D` is used. However, the element `C` in `X2` refers to a different model than `C` in `X3` and the latter is assigned the name `C_1` in the union.

4.2 Megamodel Collection Operators

In this section, we define the set of megamodel collection operators we present in this chapter: **map** – Section 4.2.1, **reduce** – Section 4.2.2 and **filter** – Section 4.2.3. Their signatures are **map**$[\mathcal{T}] : \mathscr{P}(\mathcal{M}) \rightarrow \mathscr{P}(\mathcal{M})$, **reduce**$[\mathcal{T}] : \mathcal{M} \rightarrow \mathcal{M}$, and **filter**$[\mathcal{P}] : \mathcal{M} \rightarrow \mathcal{M}$, respectively, where $\mathcal{T}$ is the set of model transformations, $\mathcal{M}$ is the set of megamodels, $\mathcal{P}$ is the set of model properties and $\mathscr{P}$ is the powerset operator. All three are higher-order operators that accept a transformation or a model property as a parameter (indicated in square brackets). We describe each operator as follows: first the standard usage, then the special adaptation needed to handle megamodels and finally, the behaviour defined as an algorithm.

4.2.1 Operator map

Standard Usage. The usual behaviour of a **map** operation is to traverse a collection (e.g., list, tree, etc.) and apply a function to the value at each node in the collection. The result is a collection with the same size and structure as the original with the function output value at each node. For example, given the list of integers $L = [10, 13, 4, 5]$ and the function *Double* that takes an integer and doubles it, applying map with *Double* to L yields the list $[20, 26, 8, 10]$. If the function has more than one argument, the mapped version can take a collection (with the same size and structure) for

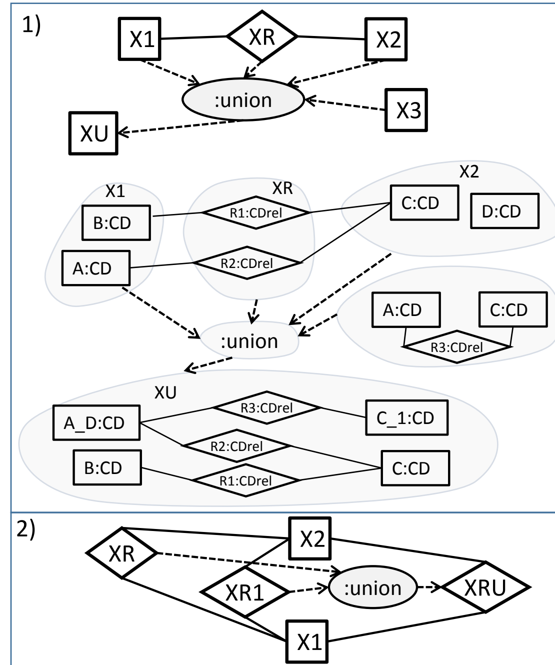


Figure 4.1: An illustration of the **union** operator applied to (1) an mgraph of megamodels (megamodel contents shown underneath), and (2) a set of megarels that share the same endpoints.

each argument, and the function is applied at a given node in the collection using the value at that node in each argument in the collection.

Adaptation for Megamodels. Since a transformation input signature is an mgraph, applying it to each node of a megamodel is not possible. Instead, the **map** operator for megamodels applies the transformation for every possible binding of the input signature in the input megamodel(s). The collection of outputs from these applications forms the output megamodel.

When the transformation signature consists of a single input and output type and uses a single input megamodel which happens to be a set (i.e., no relationships) of instances of the input type, then our **map** produces the same result as a standard map operator applied to a set. However, in the general case, **map** is more complex and differs from the behaviour of the standard map. In particular,

(1) The output megamodel may not have the same structure as the input megamodel since the structure is dependent on the output signature of the transformation.

(2) The size of the output may not be equal to the size of the input. For example, if a transformation `FF` takes two models as input and produces one as its output, applying **map** to it on a megamodel with n models will produce as many as $n \times (n-1)$ output models since each pair of input models may be matched in a binding. At the other extreme, if no input models form a binding then the output will be the empty megamodel.

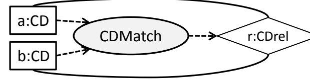


Figure 4.2: Signature of the `CDMatch` transformation.

(3) When there are multiple input megamodels, each binding of the input signature is split across the input megamodels in a user-definable way.

(4) When the transformation is commutative, we (may) want to avoid replication in the output due to isomorphic bindings. For example, if the transformation `FF` is commutative, we will get each output model twice since there are two ways to apply `FF` to a pair of models.

In what follows, we present an operator **map** for handling megamodels while avoiding the above problems.

Definition. **map**$[F](\{X_e|e \in I\})$ applies a model transformation F with a signature $\langle I, O\rangle$ to a set of input megamodels $\{X_e|e \in I\}$. Note that the megamodels X_e need not be distinct; thus multiple input arguments can be taken from the same megamodel. **map** produces an output megamodel for each element of the output signature O. The behaviour is defined by the algorithm in Figure 4.4.

We explain the algorithm using the illustration in Figure 4.3(1) of applying **map** to the `CDMatch` transformation given in Figure 4.2. The input signature consists of $\{\mathtt{a}:\mathtt{CD}, \mathtt{b}:\mathtt{CD}\}$ and the output signature is $\{\mathtt{r}(\mathtt{a},\mathtt{b}):\mathtt{CDrel}\}$. The diagram at the top of Figure 4.3 shows **map**(`CDMatch`) applied to megamodels `X1` and `X2` to produce an output megarel `XR`. Thus, the input megamodels are $X_a := \mathtt{X1}$ and $X_b := \mathtt{X2}$ and the one output, Y_r, corresponding to the output signature element `r`, produces the value for `XR` .

In line 1, the output megamodels are initialized to the empty megamodel. In our example, $Y_r = \emptyset$. Lines 2-5 iterate over all possible bindings of I in the input megamodels. In line 2, a fresh binding (i.e., previously unmatched) for the input signature of F is found in the input megamodels. Thus, in this example, a binding for `a` is drawn from `X1` and a binding for `b` – from `X2`. Assume this is $K := \{K_a := \mathtt{A}, K_b := \mathtt{C}\}$. Lines 3-4 check whether isomorphic bindings should be ignored because F is commutative. Binding isomorphisms do not occur in this example, so we illustrate them separately below. In line 5, the output of applying the transformation to the combined input binding is added to the output megamodels. Thus, in our example, `CDMatch` is applied to K and the resulting `CDrel` relationship `R2` is added to Y_r. Line 6 returns the resulting output.

In our example, there are only two matches; thus, the resulting megarel contains two relationships. However, consider the alternative application of **map** to `CDMatch` shown in Figure 4.3(2). Here both input elements are taken from the input megamodel `X`. Assume that `X` contains all three models $\{\mathtt{A}:\mathtt{CD}, \mathtt{B}:\mathtt{CD}, \mathtt{C}:\mathtt{CD}\}$. In that case, there are six possible ways to match the input signature. However, since `CDMatch` is designated as `commutative`, a binding $\{K_a := m, K_b := n\}$ produces the same output as $\{K_a := n, K_b := m\}$; thus, only three matches are used to produce the output.

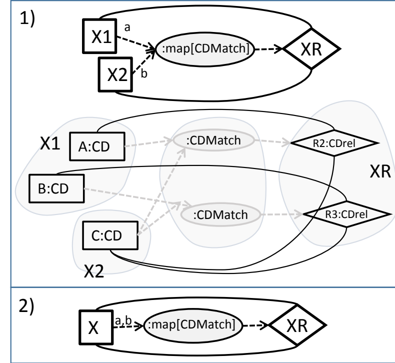


Figure 4.3: 1) An illustration of applying **map** to the `CDMatch` transformation using two input megamodels (megamodel content shown underneath); 2) using the same input megamodel for both arguments.

Algorithm: Apply map
Input: transformation F with signature $\langle I, O\rangle$,
megamodels $\{X_e | e \in I\}$
Output: set of megamodels $\{Y_e | e \in O\}$
1: **for** $(e \in O)$ { **let** $Y_e := \emptyset$ }
2: **for** (fresh binding K in $\{X_e | e \in I\}$) {
3: **if** F is `commutative` **then**
4: **if** isomorphism of K already done **then continue**;
5: **for** $(e \in O)$ { add element e of $F(K)$ to Y_e } }
6: **return** $\{Y_e | e \in O\}$

Figure 4.4: Algorithm defining behaviour of the **map** operator.

4.2.2 Operator reduce

Standard Usage. There are different variants of the reduce (also called fold, aggregate, etc.) operator used in programming languages but it typically accepts a binary function F and applies it over values $x_1, x_2, \ldots, x_n$ in a recursive collection (e.g., list, tree, etc.) by accumulating the intermediate values, e.g., $F(x_n, F(\ldots, F(x_3, F(x_2, x_1))$. For example, applying reduce with the "+" operator to the list $[1, 3, 1, 9]$ produces the sum 14.

Adaptation for Megamodels. In a similar way, we expect the **reduce** operator to accept a transformation F and use this to combine the elements of the input megamodel. Our approach is to view F as a rewrite rule, by repeatedly applying F in-place and deleting the input elements until it can no longer be applied. First, we must consider several issues:

(1) What should be the criteria that F must satisfy for this process to terminate?

(2) Since a megamodel is not a recursively defined structure and has no well-defined ordering on its elements, we cannot rely on a specific traversal path. Thus, F must be *confluent* – the final

result of **reduce** should be same regardless of the order in which we apply F to the megamodel.

(3) Since the input elements may have relationships to other neighbouring elements in the megamodel, we must be careful to preserve this information when the relationships are deleted.

We will address issues (1) and (2) in the definition of **reduce** below with appropriate assumptions on F. We address issue (3) by using relationship composition operators to construct new relationships to neighbouring elements as needed. As an illustration, assume we are using **reduce** with the `CDMerge` transformation (See Figure 2.12) to merge a megamodel of class diagrams and `CDRel` relationships. Figure 4.6 shows one iteration of the reduction. In step ❶, `CDMerge` is applied to an arbitrarily chosen pair of models (in this case, `B` and `C`) to produce a new class diagram `BC`. In step ❷, composition operators are invoked to connect `BC` to the neighbours of `B` and `C`. Finally, in step ❸, the original models `B` and `C` are deleted together with all of their relationships.

Definition. We now define a new operator **reduce**$[F](X)$ aimed to apply a transformation F to reduce the content of a megamodel X. We begin by making the following assumptions:

(I) We assume availability of predefined relationship composition operators for all relationship combinations we encounter, together with a library function `getCompOp` that provides such an operator given a pair of relationship types.

(II) In order to achieve confluence, F is required to be commutative and associative with itself and with all relationship composition operators used in item (I).

(III) In order for the reduction process to terminate, we put the constraint on F that it must be strictly reducing in output types: for every model type in the input signature, there must be fewer models of that type in the output signature; and, for relationship type in the input signature, there must be fewer relationships of that type in the output signature that are connected to output models on both (or all, for n-ary) ends.

Figure 4.5 gives the algorithm for defining the behaviour of **reduce**. In line 1, Y is initialized to the same value as the input. Lines 2-9 iterate for each binding of F in Y until no more can be found and the algorithm terminates returning Y (line 10). In the loop, for a given binding K (line 2), F is first applied to get K' line 3. Then lines 4-9 perform the steps as described in Figure 4.6 to connect the neighbours of input models in K to the output models in K' using composition operators and then deleting the input models in K. For each output model m' with a relationship r to an input model m (line 4), and for each neighbour model m'' of input model m with relationship r' (line 5), a new relationship r'' is constructed directly from m'' to m' by composing r' and r (line 7). The operator to compose a relationship of `type`(r') with one of `type`(r) must be "looked up" using `getCompOp` (line 6).

4.2.3 Operator filter

Standard Usage. Many languages provide a filtering operation to extract a portion of a collection that satisfies some condition. For example, filtering the list $[2, 5, 6, 8, 9, 1]$ using the property `isEven` produces the list $[2, 6, 8]$.

Algorithm: Apply reduce
Input: transformation F with signature $\langle I, O \rangle$,
megamodel X
Output: megamodel Y
1: **let** $Y := X$
2: **for** (binding K in Y) {
3: apply $F(K)$ generating output K';
4: **for** ($m \in K_{\texttt{Mod}}, m' \in K'_{\texttt{Mod}}, r(m, m') \in K'_{\texttt{Rel}}$) {
5: **for** ($m'' \in Y_{\texttt{Mod}}, r'(m'', m) \in Y_{\texttt{Rel}}$) {
6: **let** $comp :=$ `getCompOp`(`type`(r'), `type`(r));
7: **let** $r''(m', m'') := comp(r', r)$;
8: add r'' to Y }}
9: delete elements in K from Y }
10: **return** Y

Figure 4.5: Algorithm defining behaviour of the **reduce** operator.

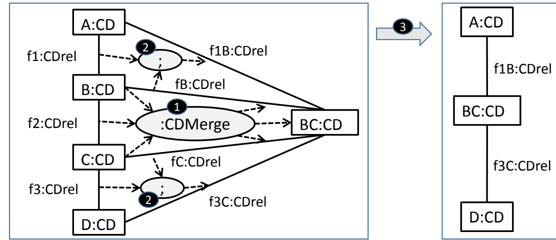


Figure 4.6: An illustration of one iteration of **reduce**. First the merge is applied non-deterministically (step 1). Then the relationships to the neighbours of the merged models are computed using appropriate composition operators. Finally, all input elements are deleted.

Adaptation for Megamodels. The **filter** operator is similar and applies to megamodels. A property is given as the filtering condition, and the subset of elements that satisfy the property is used to produce the output. We distinguish between model and relationship properties and treat them independently. Thus, a *model property* filters only models and keeps all relationships between the remaining models. A *relationship property* filters only relationships and does not affect the models.

filter differs from **map** and **reduce** in that it does not create new models or relationships; it just creates new references to existing models and relationships. Thus, all elements of the output megamodel refer to artifacts that are already referred to by elements of the input megamodel. This aspect of **filter** makes it an inexpensive operation compared with **map** or **reduce**.

If a property P, defined for a model or relationship type T, is used for **filter**, then it selects all elements of type T (or its compatible types) that satisfy the constraints in P (See Definition 12). It is also possible to give a type T as the property which is interpreted as the property *true*, satisfied by any instance of T (or its compatible types).

Algorithm: Apply filter
Input: property P, megamodel X
Output: megamodel Y
1: **let** $Y := \emptyset$;
2: **for** $(m \in X_{\texttt{Mod}})$ {
3: **if** P is a model property **then**
4: **if** $m \models P$ **then** add m to Y;
5: **else** add m to Y }
6: **for** $(r \in X_{\texttt{Rel}})$ {
7: **if** P is a relationship property **then**
8: **if** $r \models P$ **then** add r to Y;
9: **else if** $r.\texttt{end} \cap Y \neq \emptyset$ **then** add r to Y }
10: **return** Y;

Figure 4.7: Algorithm defining behaviour of the **filter** operator.

Definition. **filter**$[P](X)$ filters megamodel X to produce the least sub-megamodel of X containing all the elements of X that satisfy property P.

The behaviour of **filter** is given by the algorithm in Figure 4.7. Line 1 initializes the output to the empty megamodel. Lines 2-5, iterate over the model elements in X. If P is a model property then the model is only added to the output if passes the satisfaction check (line 4). If P is not a model property, all models are added to the output (line 5). A similar algorithm is followed in lines 6-9 that iterate over relationship elements. The only difference is that if P is not a relationship property (and so it must be model property), only those relationships are added to the output that already have their endpoints in the output due to the filtering in lines 2-5.

4.3 Application Scenarios

In this section, we illustrate our collection-based megamodel management operators using several scenarios.

4.3.1 Experiment Driver

The goal of this scenario is to apply a transformation on a megamodel and perform some kind of experiment on the result of its application. Specifically, given a megamodel `XCD` containing a set of class diagrams, we wish to apply transformation `CD2Java` that translates a class diagram to its equivalent Java code and produces a `CD2JavaRel` traceability relationship from the CD to the Java code. Then, we wish to apply evaluation transformation `ECheck` on each `CD2JavaRel` in the megarel resulting from the transformation application. `ECheck` computes the number of classes in each transformed class diagram that do not have Java counterparts. Finally, we would like to sum these up via a `Sum` operation to learn the total number of incidents where this occurs. If this is

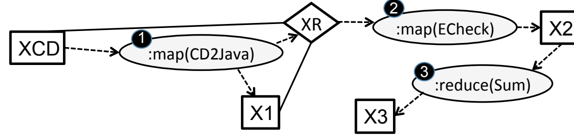


Figure 4.8: Experiment driver scenario illustration.

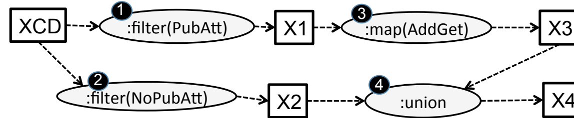


Figure 4.9: Mass refactoring scenario illustration.

greater than zero, then we will identify a problem in the transformation. Figure 4.8 shows the chain of operators required to accomplish this via the following steps:

(1) Apply **map**[CD2Java](XCD) to produce X1 which contains the Java code and XR which is the megarel containing all relations between XCD and X1.

(2) Apply **map**[ECheck](XR) to produce megamodel X2 which contains the evaluation ECheck for each rel in XR.

(3) Apply **reduce**[Sum](X2) to produce the final result X3 containing a single value which is the sum of the results of **map**[ECheck](XR). A value is greater than zero indicates that there was a problem in the transformation application.

4.3.2 Mass Refactoring

We are given a megamodel XCD that contains unrelated class diagrams, a property PubAtt that represents models with public attributes and its negation NoPubAtt. We wish to find models satisfying PubAtt and refactor them so that public attributes become private attributes with public getter methods using refactoring transformation PubGet. Figure 4.9 illustrates this scenario via the following steps:

(1) Apply **filter**[PubAtt](XCD) to produce a megamodel X1 containing the sub-megamodel of XCD with models where property PubAtt holds.

(2) Apply **filter**[NoPubAtt](XCD) to produce a megamodel X2 containing the sub-megamodel of XCD with models where property PubAtt does not hold.

(3) Apply **map**[AddGet](X1) to transform the models with the undesirable property using a refactoring transformation AddGet which produces a megamodel X3.

(4) Return a megamodel X4 = **union**(X2, X3) (see Section 4.1) which represents the refactored version of the original s.t. the property PubAtt no longer holds on any of its models.

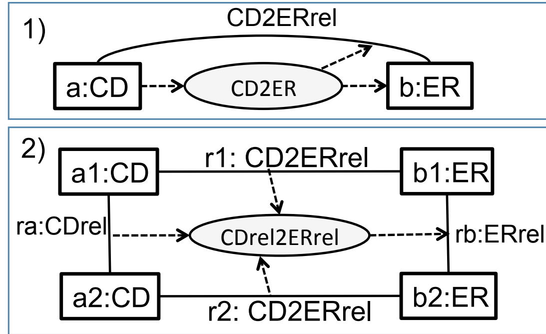


Figure 4.10: Illustration of transformation signatures for megamodel transformation scenario. (1) Class Diagram (CD) to Entity Relationship (ER) transformation, (2) CD relation to ER relation transformation.

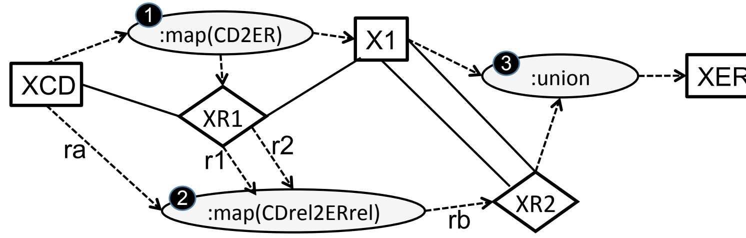


Figure 4.11: Megamodel transformation scenario illustration.

4.3.3 Megamodel Transformation

We are given an input megamodel `XCD` consisting of class diagrams (`CDs`) related by class diagram relations (`CDrels`), and we wish to transform it to a megamodel `XER` consisting of ER diagrams (`ERs`) related by ER diagram relations (`ERrels`). We are also given the transformations `CD2ER` and `CDrel2ERrel` (See signatures in Figure 4.10) which transform `CDs` to `ERs` and `CDrels` to `ERrels`, respectively. We would like to use our operators to accomplish this.

The steps to perform this transformation are illustrated in Figure 4.11 and involve the following steps:

(1) Apply **map**[`CD2ER`](`XCD`) which based on its signature applies only to the (`CDs`) in `XCD` and produces the megamodel `X1` consisting of the `ER` versions of all the `CDs` in `XCD` as well as the megamodel relation `XR1`.

(2) Apply **map**[`CDrel2ERrel`](`XCD`) which based on its signature applies only to the `CDrels` in `XCD` and produces the megamodel relation `XR2` consisting of a set of `ERrels` with endpoints in `X1`. Note that the other arguments come from the megamoodel relation `XR1` which contains the applications of the `CD2ER` transformation.

(3) Apply **union**(`X1`, `R`) to produce the final megamodel `XER` which contains the corresponding `ERs` and the `ERrels` between them.

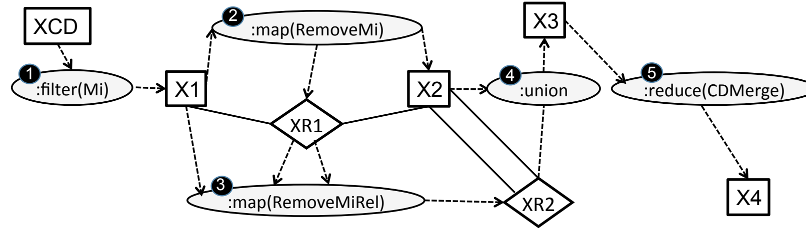


Figure 4.12: Motivating example illustration.

4.3.4 Scenario from the Motivating Example

Recall the motivating scenario: given a megamodel `XCD` which contains class diagrams and an undesirable property `Mi` that represents class diagrams with multiple inheritance, we aim to identify all models that are of type class diagram and contain this property, refactor them using a predefined transformation to remove the property, and merge the modified class diagrams. Figure 4.12 shows the chain of operators required to accomplish this scenario:

(1) Apply **filter**[`Mi`](`XCD`)to produce a megamodel `X1` containing the sub-megamodel of `XCD` with models where property `Mi` holds.

(2) Based on the megamodel transformation pattern described in Scenario C: apply **map**[`RemoveMi`](`X1`) to produce `X2` which is the refactored version of `X1` that no longer contains the undesirable property, and (3) apply **map**[`RemoveMiRel`](`XR1`) to produce the megamodel `XR2` containing the relations between the refactored models.

(4) Apply **union**(`X2`, `XR2`) to produce `X3` which is the megamodel containing the refactored models and relations between them.

(5) Apply **reduce**[`CDMerge`](`X3`) which applies the `CDMerge` operation described in Section 4.2 on class diagrams with relation `CDRel` between them and produces a megamodel `X4` where all the related class diagrams are now combined. The final result can now be compared with the result of merging the pre-refactored models which can be achieved by using **reduce**[`CDMerge`](`XCD`).

Although the scenarios we have presented address specific types of megamodels, transformations and properties, they can be generalized as design patterns for similar reoccurring problems. For example, the mass refactoring scenario can be generalized for any problem that involves a megamodel which may contain elements with a certain property which should be removed. Similarly, the megamodel transformation scenario can be generalized for any problem that involves a transformation of one type of megamodel to another, given the appropriate transformations between source and target models and source and target relations. We have observed that in the case of the megamodel transformation pattern, the relation transformation can be induced from the model transformation; however, further analysis is outside the scope of our work.

map$[F](\{X\})$	$O(n^k \times C_F(m))$
reduce$[F](X)$	$O(n^2 \times C_F(m))$
filter$[P](X)$	$O(n^q \times C_P(m))$

Figure 4.13: Worst case complexity of the operators.

4.4 Analysis

In this section, we analyze the worst case complexity of the three operators presented – see the summary in Figure 4.13 – and discuss the implications of this for scalability.

Complexity of map. For the algorithm in Figure 4.4, the iteration in line 2 over possible bindings of input signature I can execute up to n^k times, where n is the number of models input megamodels and k, the number of models in I. If F is commutative with q isomorphisms of I, the loop can execute $(n^k)/q$ times. Thus, the complexity is $O(n^k \times C_F(m))$ where $C_F(m)$ is the complexity of executing F in terms of a size metric m of the input binding to I. We assume that there is at most one relationship of each type between any given set of models in input megamodels.

Complexity of reduce. Line 2 of the algorithm in Figure 4.5 iterates over all possible bindings of input signature I, but each time, the input models and relationships are deleted. Thus, each input element participates in at most one binding. Furthermore, due to the assumption that F is strictly reducing, each iteration reduces the number of models and relationships. Thus, the number of iterations is bounded by n, the number of models in X – as with **map** we assume there is at most one relationship of each type between a given set of models. The internal loops lines 4-8 iterate once for every neighbour of a model M in I and relationship of M to a model in O. This can iterate up to rn times where r is the number of relationships between O and I. Since r is a constant for a given F, the complexity is $O(n^2 \times C_F(m))$.

Complexity of filter. For a property, the algorithm in Figure 4.7 iterates n times while for a relationship property over a q-ary relationship, it iterates wn^q times, where n is the number of models in X and w is the number of q-ary relationship types. Since we assume w is a constant, the complexity is $O(n^q \times C_P(m))$ where $C_P(m)$ is the complexity of checking P in terms of a size metric m of the input relation.

Discussion. The analysis results in Figure 4.13 show that the operators scale reasonably for certain classes of application scenarios. Specifically, the complexity is no worse than quadratic (modulo the transformation/property complexity) in the size of the input megamodel when **map** is applied to a transformation with two or fewer input models, in all cases for **reduce** and when **filter** is applied to either a model property or to a binary relationship property. Some scenarios exceed these limits (e.g., scenario C in Section 4.3); we plan to address scalability issues in future work.

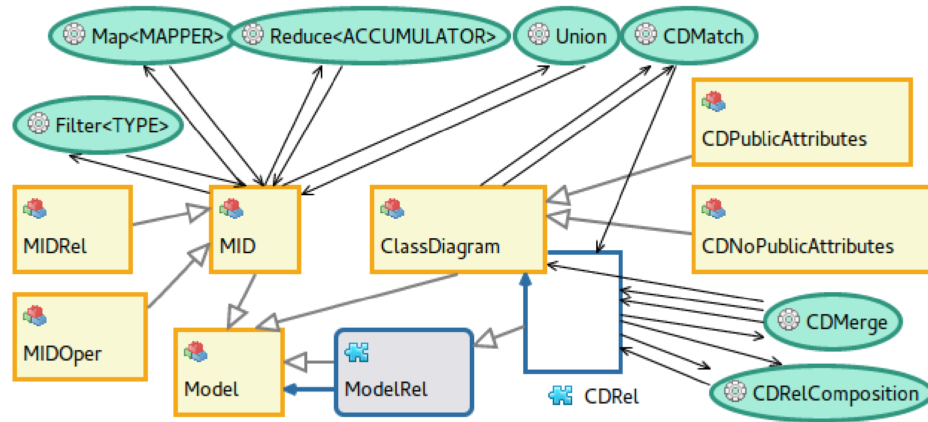


Figure 4.14: Type megamodel in *MMINT* used for the examples in this chapter.

4.5 Tool Support

4.5.1 Using *MMINT*

The megamodel collection operators described in this section have been implemented in the *MMINT* tool described in Chapter 3.

Recall that *MMINT* uses a distinguished *type megamodel* in which model types, relationship types and transformations are registered. Figure 4.14 shows a screenshot of the type megamodel used to implement examples in this section. Here, boxes represent model types and links between them are binary relationship types (thick blue arrows). The sub-typing between types is shown with the hollow-headed arrows. Transformations are ovals connected to their input and output types with named links (names are not shown to avoid clutter). The transformation signature information can be extracted directly from this model. Additional metadata such as whether a transformation is commutative or is a relationship composition relation is also stored in this model.

The runtime operation of *MMINT* is centred around a megamodel editor that allows an engineer to interactively create models and relationships, invoke transformations on them and inspect the results. Implementations for supporting tools such as type-specific editors, validation checkers, solvers and custom transformation implementations can be plugged in and are managed by the type support runtime layer. A generic relationship editor is built into the *MMINT* tool.

4.5.2 Implementation of Collection Operators

In *MMINT*, a megamodel is referred to as a `MID` (Model Interconnection Diagram). All transformations, including higher-order ones, are registered in the type megamodel. Thus, the three collection megamodel operators in this chapter can be seen at the top of Figure 4.14 with their inputs and outputs connected to the `MID` type indicating that they take megamodels as input and produce them as output. In addition, each accepts a parameter (within the angle brackets). **union** can also be

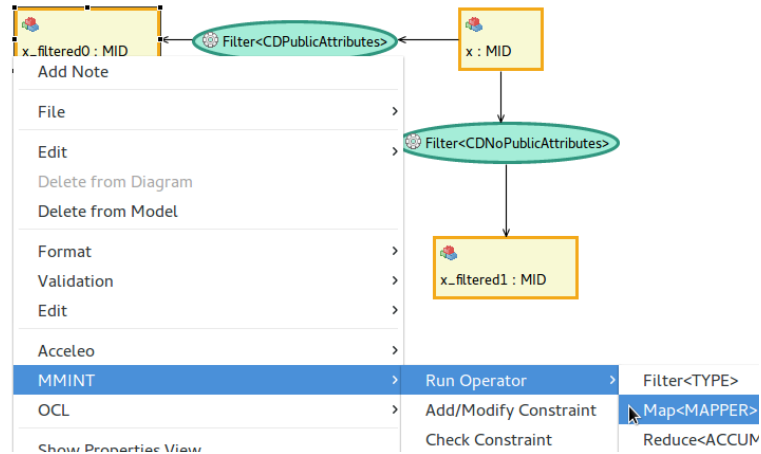


Figure 4.15: Screenshot of megamodel for scenario B in Section 4.3 being built in the *MMINT* megamodel editor.

seen as an unparameterized transformation. The type compatibility relation (see `TypeComp` from Definition 7) is given by the subtype relation in the type megamodel.

Properties are implemented in *MMINT* as a model or relationship subtype that contains additional well-formedness constraints but does not change the metamodel of its supertype. For example, the `CDPublicAttributes` type is used for scenario B in Section 4.3 and contains the following OCL code:

```
CDPublicAttributes:
  Attribute.allInstances()->exists(
    attribute | attribute.public)
```

The algorithms in Figure 4.4, 4.5 and 4.7 for the three operators (and **union**) are implemented in Java and plugged into the type support runtime layer as transformation definitions. At runtime, when an engineer selects one or more MID elements and right-clicks to see what transformations are available to apply, they see these operators and can select one to apply. If it is parameterized, then a second dialog appears showing the choices for the parameter. For example, Figure 4.15 shows a screenshot of scenario B in Section 4.3 being built in the megamodel editor. Currently, the engineer is on step 3 and is invoking the **map** operator.

4.5.3 Experiments

The *MMINT* implementation was used to express each of the four scenarios described in Section 4.3. Although these are "toy" experiments, they exercise the different aspects of the implementation. As a preliminary robustness test of the implementation, we ran an additional experiment in which we populated a megamodel with a varying number of class diagrams. The class diagrams were generated to contain 5 distinct random classes, picked from a pool of 50 available classes. Thus,

	# CDs	# rels	time (sec)	MID size (MB)
exp1	10	100	0.15	0.2
exp2	100	10000	12.92	20.7
exp3	250	62500	85.74	128.9
exp4	500	250000	422.78	518.7

Table 4.1: Experimental results running **map**[`CDMatch`]

many of the class diagrams share classes with the same name. We then measured the running time of **map**[`CDMatch`], given that **map** is the most complex of the operators, and the memory size of its output. The results are shown in Table 4.1.

Since `CDMatch` has two input models, the complexity formula in Figure 4.13 predicts quadratic time in the number of models of the input megamodel. The results in Table 4.1 are consistent with this prediction and additionally show that the space also increases quadratically. Although 422s (∼7min) does not seem excessive to process 500 models, we plan to improve scalability further.

4.6 Related Work

Many model management approaches have been proposed. For example, Rondo [Melnik *et al.*(2003)] represents models as directed labeled graphs and supports traditional model management operations (e.g., match and merge) that work directly on models but not on the megamodels containing them. Maudeling[1] offers advanced query services; however, these are on the modeling artifacts themselves and not on megamodels. Epsilon [Kolovos *et al.*(2015)] provides a set of domain specific languages for specific model management operations such as match and merge; however, no special support is provided for megamodels.

The Atlas Model Management Architecture (AMMA) [Bézivin *et al.*(2005a)] has a component AM3 for expressing megamodels and an OCL-based scripting language MoScript for general model management scripts including limited support for megamodel manipulation. Specifically, MoScript [Kling *et al.*(2012)] provides support for *map* by using the OCL ApplyTo and Collect operations and support for *filter* using the OCL Select operation; however, these versions of map and filter are more limited than what we presented because MoScript does not treat relationships between models as first class citizens and the support for map and filter is limited to sets of models rather than graph-like collections in megamodels. In addition, MoScript does not provide support for the reduce operation. Despite these weaknesses, we see MDE workflow languages such as MoScript, UniTI [Vanhooff *et al.*(2007)], and TraCo [Heidenreich *et al.*(2011)] as complementary to our approach and believe they can benefit from incorporating our megamodel manipulation operations into the language. We leave the investigation of such integration for future work.

Model search engines such as MOOGLE [Lucrédio *et al.*(2008)] or IncQuery [Ujhelyi *et al.*(2015)]

[1] `Maudeling:http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling`

perform queries of model contents. Our **filter** operation does not limit which languages or engines can be used for defining model and relationship properties. Thus, model search engines are complementary to our approach.

Graph-based languages and frameworks that provide collection-based operations on graphs have been proposed. The map and fold (i.e., reduce) algorithms in [Erwig(1997)] generalize the classic list-based versions of these to graphs but the assumptions made by these algorithms make them inapplicable to the megamodel case. Specifically, the map algorithm does not allow for a "graph" of input arguments to the transformation as **map** does with transformation input signatures, and the fold algorithm only aggregates values on nodes and edges rather than collapsing the graph structure itself as **reduce** does. The MapReduce approaches of Google and others [Dean and Ghemawat(2008)] are intended for the efficient processing of big data; yet these operate differently from the *map* and *reduce* functions found in many programming languages [Lämmel(2008)].

4.7 Chapter Summary

In this chapter, we presented three new megamodel collection operators: **map**, **reduce**, and **filter**. These operators are inspired by similar collection manipulation operators found in many programming languages, but are adapted to address the special characteristics of megamodels and MDE environments. Specifically, the operators treat model relationships as first class entities and address the graph-like structure of megamodels and of the signatures for model transformations.

Chapter 5

Heterogeneous Megamodel Slicing

Slicing is a widely used technique for supporting comprehension and assessing change impact during software evolution activities. While there has been substantial research into the slicing of particular model types, model-based software development typically involves heterogeneous collections of related models and there is little work addressing slicing in this context. In this chapter, we propose a generic slicing approach for megamodels. Our approach exploits existing model slicers for particular model types as well as the traceability relationships between models to address the broader heterogeneous model slicing problem. We illustrate our approach on an example of evolution in model-based automotive software development using the PSD system.

Content in this chapter was published in [Salay *et al.*(2016)], where I contributed to defining the megamodel slicing algorithm and applying it on a worked out example, as well as surveying the literature for related work.

Introduction

Slicing is a widely used technique for supporting software evolution activities [Li *et al.*(2013)]. Specifically, *static* slicing [Weiser(1981)] can identify the subset of software that is semantically dependent on a specific portion that has or is planned to be changed and hence is useful for assessing change impact due to evolution. In the MDE context, model slicing has been studied for particular model types, e.g., State Machines [Korel *et al.*(2003), Lano and Rahimi(2010)], Class Diagrams [Kagdi *et al.*(2005), Lano and Rahimi(2010)], etc. However, large-scale software systems are often described using heterogeneous collections of interrelated models, and change impact analysis requires a broader slicing approach that can address this.

While some work has addressed slicing for heterogeneous model collections, these have been limited to a specific set of model types (e.g., [Nejati *et al.*(2012)]) or remain at a theoretical level (e.g., [Clark(2011)]).

Contributions. In this chapter, we present a general and pragmatic static slicing algorithm for heterogeneous model collections. Specifically, (1) it operates on "megamodels" – a general modeling technique to represent collections of interrelated models; (2) it can work with arbitrary model types (e.g., conceptual, behavioural, goal models, test models, etc.) by utilizing their corresponding type-specific model slicers; and (3) it uses the widely adopted notion of traceability relations to assess change impact between models. We then analyze the proposed algorithm for termination, correctness, running time and minimality.

Organization. The remainder of this chapter is structured as follows. In Section 5.1, we give a motivating example. In Section 5.2, we describe the proposed slicing algorithm and its analysis. Then, in Section 5.4, we give a detailed illustration of the algorithm on the PSD example. In Section 5.5, we discuss related work and finally, in Section 5.6, we give a chapter summary.

5.1 Motivating Example

Consider that the power sliding door system introduced in Section 2.1 changes and the redundant switch is removed. This could be due to the need to minimize cost and produce a cheaper vehicle. In the new system, only the AC ECU checks the vehicle speed before commanding Actuator. In this case, it would be desirable to provide a sliced megamodel of the system that reflects the parts of the original megamodel affected by this change in order to help with system evolution activities. For example, we show in Chapter 7, that with safety-critical software, such as for automotive systems, a system megamodel slice is an essential part of re-assessing the safety assurance of the system. We demonstrate our slicing approach on the PSD example in Section 5.4.

5.2 Megamodel Slicing

In this section, we present a slicing approach for heterogeneous megamodels. Intuitively, such a slicer should allow the criterion to be expressed as a megamodel fragment and the forward slice should expand this to the megamodel fragment containing all dependent elements. We generalize Definition 17 to capture this intuition.

Definition 18 (Static forward megamodel slice) *Given a megamodel X and megamodel fragment $S[X]$, the* static forward slice *of X with respect to slicing criterion $S[X]$ is the megamodel fragment $S'[X]$ satisfying the following conditions for all $M \in X$:*

1. *(Correctness) There exists a model fragment $S'[M] \in S'[X]$ that contains all atoms of M that are directly or indirectly dependent on the atoms of any model fragment in $S[X]$.*

2. *(Minimality) If there exists a model fragment $S'[M] \in S'[X]$, then every atom of $S'[M]$ is either directly or indirectly dependent on the atoms of some model fragment in $S[X]$.*

There are two levels of expansion in this slicing process: (1) expansion within individual models to dependent elements and, (2) expansion between models across relationships to dependent elements in neighbouring models. This two-level process is repeated until it produces no further expansion. For (1), we leverage existing type-specific slicers that take the semantics of the individual model types into account. For (2), we use the links in traceability relationships to connect dependent elements. Here, no special relationship-type specific slicers are needed since all relationship types are assumed to be sets of links and every link is assumed to represent a dependency.

Note that this definition of slicing is a *deep* slicing since the process includes the content of the models and relationships referenced by the elements of the megamodel. In contrast, a *shallow* megamodel slicing would be one that only considered the elements of the megamodel and not what they reference. Here, a subset of a megamodel (the criterion) is expanded to the subset that is connected directly or indirectly via relationship links (the slice), i.e., the shallow slice is the largest subset contained in the transitive closure of the initial subset taken along relationship links. There may be some use cases in which shallow megamodel slicing is useful but in this chapter we focus on the deep version.

5.2.1 Slicing algorithm

Figure 5.1 gives the algorithm for forward slice. The input is megamodel X with megamodel fragment $S_c[X]$ given as the slicing criterion. The output is megamodel fragment $S[X]$ representing the forward slice. The algorithm makes the following assumptions:

Assumption 1 *For each model type T represented in X, we have a slicer* $\texttt{Slice}_T$ *for models of type T that satisfies Definition 17.*

Assumption 2 *The set of traceability relationships in X express all and only the direct dependencies between atoms of models in X.*

In addition, we require several simple supporting operations.

Definition 19 (Union) *Given a pair of megamodel fragments $S_1[X], S_2[X]$, the* megamodel fragment union*, denoted* $\texttt{Union}(S_1[X], S_2[X])$*, is defined with the following condition.*

$\forall S[M] \in \texttt{Union}(S_1[X], S_2[X]) \cdot$

$$S[M] = \cup\{S'[M] | S'[M] \in S_1[X] \cup S_2[X]\}$$

Thus, the $\texttt{Union}(S_1[X], S_2[X])$ can be constructed by first taking the set union $S_1[X] \cup S_2[X]$ and then unioning all model fragments of the same model within this.

Definition 20 (Trace) *Given a traceability relationship R with ends M and M', and model fragment $S[M]$, the* trace *of $S[M]$ along R, denoted* $\texttt{Trace}(R, S[M])$ *is the model fragment $S'[M']$ consisting of the subset of atoms in M' dependent on the atoms in M according to R.*

We compute $\texttt{Trace}(R, S[M])$ by following the links of R from the atoms of M to the atoms of M'.

Definition 21 (OppEnd) *Given a traceability relationship R with ends M and M', we define* $\texttt{OppEnd}(R, M) = M'$ *and* $\texttt{OppEnd}(R, M') = M$.

In line 1 of the algorithm, the current slice is initialized to the criterion. The two levels of expansion are in lines 4-9 and lines 10-17, respectively, inside the main loop of lines 2-19. For level 1, the temporary result $S_1[X]$ is initialized in line 3 to the empty set and then lines 5-9 iterate through the model fragments in the current slice. In line 7, the model type-specific slice is computed using the model fragment as the criterion and the result is accumulated in $S_1[X]$ (line 8).

The level 2 expansion temporary result $S_2[X]$ initialized on line 10. The outer iteration (lines 11-17) is over the model fragments from the level 1 expansion, and the inner iteration (lines 12-16) is over each relationship connected to the model fragment. Note that the set of relationships connected to a model fragment $S_1[M]$ is the set of relationships connected to M via the `end` property (see Figure 2.8). For each such relationship R, we first determine the model M' on the other end of the relationship using supporting function `OppEnd` in line 13. Then in line 14, the model fragment $S_2[M']$ is produced by tracing the links in R from $S_1[M]$ to M'. Finally, in line 15, this result is accumulated in $S_2[X]$.

After the two levels of expansion, the combined result is computed in line 18 and checked to see if any actual expansion has occurred (line 19). If no expansion has occurred, a fixed point has been reached and the main loop exits with the current slice returned as the final result in line 20; otherwise, the main loop repeats.

5.2.2 Analysis

We consider the issues of termination, complexity and correctness for forward slice algorithm in Figure 5.1.

Termination. We show that the slicing algorithm is guaranteed to terminate. After the level 1 expansion loop completes (lines 5-9), it is clear that $S'[X] \sqsubseteq S_1[X]$ since $S_1[X]$ is constructed by expanding each model fragment in the current slice $S[X]$ using type-specific slicers (see Assumption 1) and doing Union (see Definition 19). Furthermore, $S'[X] = S[X]$ (line 3). Then, in line 18, when the new slice is computed, $S_1[X] \sqsubseteq S[X]$ since Union cannot produce a result smaller than its arguments. Therefore, $S'[X] \sqsubseteq S[X]$. Thus, on line 19, either no expansion has occurred ($S[X] \sqsubseteq S'[X]$) and the algorithm terminates or some expansion has occurred and the loop iterates again. Thus, in each iteration, the current slice can only get larger and since this process is bounded by X, the algorithm must terminate.

Time Complexity. The level 1 loop (lines 5-9) can iterate N_M times and the level 2 loop (lines 11-17) can iterate N_M^2 times where N_M is the number of models in X. The dominating operation in the level 1 loop is the type-specific slicer. Since the time complexity varies according to the slicer

Algorithm: Forward Megamodel Slice
Input: megamodel X, criterion megamodel fragment $S_c[X]$
Output: slice megamodel fragment $S[X]$

1: $S[X] := S_c[X]$
2: **do** {
3: $S'[X] := S[X]$
4: $S_1[X] := \emptyset$
5: **for** $(S[M] \in S[X])$ {
6: $T := M.\texttt{type}$
7: $S_1[M] := \texttt{Slice}_T(M, S[M])$
8: $S_1[X] := \texttt{Union}(S_1[X], \{S_1[M]\})$
9: }
10: $S_2[X] := \emptyset$
11: **for** $(S_1[M] \in S_1[X])$ {
12: **for** $(R \in M.\texttt{end})$ {
13: $M' := \texttt{OppEnd}(R, M)$
14: $S_2[M'] := \texttt{Trace}(R, S_1[M])$
15: $S_2[X] := \texttt{Union}(S_2[X], \{S_2[M']\})$
16: }
17: }
18: $S[X] := \texttt{Union}(S_1[X], S_2[X])$
19: } **until** $(S[X] \sqsubseteq S'[X])$
20: **return** $S[X]$

Figure 5.1: Algorithm for forward megamodel slice.

used, we represent it using a type-independent upper bound $SL(n)$ as a function of the number of elements n in the input model. Tracing along a relationship and union (lines 14-15) is $O(N_a)$ in the worst case, where N_a is the total number of atoms across all models of X. Thus, in the worst case, one iteration of the main loop is $O(N_M \times SL(N_a) + N_M^2 \times N_a)$. Finally, in the worst case, the size of the current slice can increase by one in each iteration of the main loop, for N_a iterations. Thus, the time complexity is given by:

$$O(N_a \times N_M \times SL(N_a) + N_M^2 \times N_a^2)$$

Correctness. We argue that the slicing algorithm satisfies the correctness condition in Definition 18. Assume that the algorithm is at line 3 and there exists a non-empty set of atoms not in the current slice $S[X]$ that are dependent on atoms of model fragments in $S[X]$. Note that if some atom a is indirectly dependent on an atom a', then there must be a sequence of directly dependent atoms $a, a_1, ..., a_n, a'$ connecting them. Thus, there must also be a non-empty set of atoms not in $S[X]$ that are *directly* dependent on atoms of model fragments in $S[X]$. Let us choose one such atom a' in some model M' in X that is directly dependent on an atom a in some model fragment $S[M]$ in $S[X]$. We consider the two cases: $M' = M$ and $M' \neq M$.

Case 1). If $M' = M$, then by Assumption 1, the slicer used in line 7 satisfies the correctness condition in Definition 17 and thus, atom a' will be added to a model fragment in $S_1[X]$ in an iteration of the level 1 loop (lines 5-9).

Case 2). If $M' \neq M$, then by Assumption 2, there is a traceability relationship R in X with a link that connects a to a' and thus, line 14 will cause a' to be added to a model fragment in $S_2[X]$ in an iteration of the level 2 loop (lines 11-17).

In either case, the atom a' will enter the next iteration of the slice in line 18. Furthermore, since the addition of a' expands the slice, the main loop will iterate again and will capture the next set of directly dependent atoms, and so on. When the set of directly dependent atoms not in $S[X]$ is empty, no further level 1 or level 2 expansion is possible, and the algorithm terminates.

Minimality. We show that the slicing algorithm satisfies the minimality condition in Definition 18. To do this, we must show that the slice produced by the algorithm contains no atom that is not dependent on the criterion. Assume that there is an atom a' in the final slice that is not dependent on the criterion. In this case, a' must have been added to the slice on line 7 or line 14 in some iteration of the main loop. However, by Assumption 1 and Definition 17, $\texttt{Slice}_T$ can only produce minimal model slices in line 7 and so a' could not have been added there. Also, by Assumption 2, traceability relationships only contain links between true dependencies and in line 14, Trace is applied from the current slice to these dependent atoms. Thus, a' could not have been added at line 14. Therefore, we have a contradiction and so the megamodel slice must be minimal.

5.2.3 Discussion

Well-formedness and referential integrity. Definition 17 does not require that a slice be a well-formed model. However, in practice, ensuring that a slice is well-formed may be desirable because the slice can be used directly by tools such as editors, analyzers and transformations. Making a model fragment into a well-formed model requires it to be expanded by a minimum number of atoms in order to satisfy the well-formedness constraints. For example, if a CD fragment contains an association without one of its endpoints, adding the missing endpoint class will make it well-formed.

The problem with doing this expansion is that atoms can be added that are *not dependent on the criterion* since, if they were dependent, then they would already be in the slice. In particular, if $\texttt{Slice}_T$ used in line 7 of the slicing algorithm always included an expansion to well-formedness then in the subsequent steps of the algorithm the atoms added for well-formedness would be treated as though they were atoms added for dependency. This would result in a non-minimal megamodel slice. As a result, we view the expansion to well-formedness as an *optional post-processing step* that could be applied after the megamodel slice is computed.

A similar argument can be made about the issue of *referential integrity*. Assume that one atom references another, e.g., a lifeline in a sequence diagram references the class of the object that the lifeline represents. The referenced class is not dependent on the referencing lifeline; thus, if the forward slice includes the lifeline, it need not contain the class. However, it may be desirable to

expand the slice to include the class to provide relevant contextual information for the lifeline. As with well-formedness, this referential integrity expansion can introduce atoms that are not dependent on the criterion and thus such an expansion should only be done as a post-processing step on the slice.

Generalizing the slicing algorithm. In Section 2.3, we made several simplifying assumptions in order to focus on the core aspects of the slicing algorithm. We now briefly discuss how to relax these assumptions.

• **N-ary Relationships.** We have assumed that all relationships in the megamodel are binary but it is straightforward to extend the algorithm to handle N-ary relationships. Specifically, the iteration through the relationships (lines 12-16) must be generalized to handle the case where a traceability link holds between atoms in models on multiple `ends`, and the supporting operations `OppEnd` and `Trace` must be adapted to address this.

• **Nested megamodels.** In the general case, a megamodel can contain other megamodels. Such a megamodel could be viewed as a tree with models as leaves and nested megamodels as intermediate nodes. A megamodel fragment is a tree with the same structure but with model fragments as leaves. Thus, the algorithm follows a similar approach as currently but in addition it must preserve the megamodel tree structure in the final slice.

• **Arbitrary relationships.** We have assumed that all relationships are traceability relationships since these are the only ones that matter to the slicing algorithm. In general, however, there may be other types of relationships in the megamodel, e.g., refinement, overlap, etc. The simplest way to allow these relationship types is to ignore all non-traceability relationships in the loop in lines 12-16.

5.3 Megamodel Slicing with Collection-Based Operators

So far, we have proposed a generic megamodel slicing approach and gave the algorithm using traditional pseudo code. Here, we show how to re-implement the algorithm using collection operators from Chapter 4.

Figure 5.2(a) gives the general signature of a polymorphic model Slice operator, where sc is the criterion expressed as a submodel of model m. The output slice sl is another submodel of m. In this context, we use the unary relationship type Sub for expressing submodels of a model. A Sub relationship connected to a model m contains a set of links, each of which connects to an element of m; thus, it identifies a subset of elements in m. The definition of *dependent on* clearly varies according to the model type; thus, Slice is a natural example of ad hoc polymorphism.

For megamodel slicing, we assume that a *sub-megamodel* is any set of submodels from a subset of models in the megamodel. For example, Figure 5.3(a) shows a megamodel and Figure 5.3(b) – a sub-megamodel consisting of submodels shown by the shaded ovals. The original megamodel is depicted using dashed lines for clarity but the sub-megamodel consists only of the submodels. Similarly to

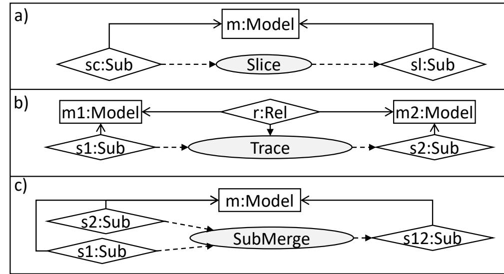


Figure 5.2: Signature of polymorphic operators required in the megamodel slicing scenario: (a) Slice; (b) Trace; and (c) SubMerge.

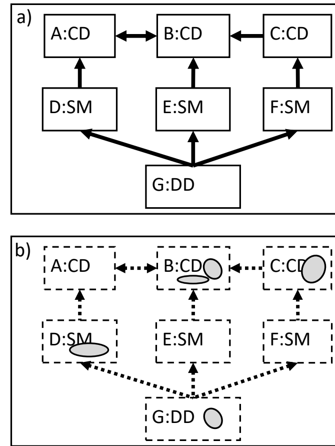


Figure 5.3: (a) An example megamodel and (b) its sub-megamodel.

submodels, we represent a sub-megamodel of a megamodel X as a unary megaRel connected to X and consisting of a set of Sub relationships connected to the models within X.

To slice a megamodel, it is not sufficient to just **map** polymorphic Slice over all models in the megamodel because we must take into account the fact that there may be inter-model dependencies across the relationships that connect models. To address these, we assume that we have a polymorphic Trace transformation with signature as shown in Figure 5.2(b) that takes a submodel of model m1 and propagates it to the dependent submodel of model m2 across the connecting relationship r. Finally, because using Slice and Trace can produce multiple submodels for the same model (i.e., one from Slice and zero or more from propagating using Trace for each neighbouring model), we also assume that we have a polymorphic submodel merge operation to combine the submodels into a single submodel – this is provided by the SubMerge transformation with the signature in Figure 5.2(c).

Figure 5.4 shows the core steps of the megamodel slicing operation. The slice sub-megamodel of a megamodel X is obtained by repeating the following four steps starting with XSin as the criterion sub-megamodel and then assigning XSin:=XSout until there is no further change (i.e. a fixed point

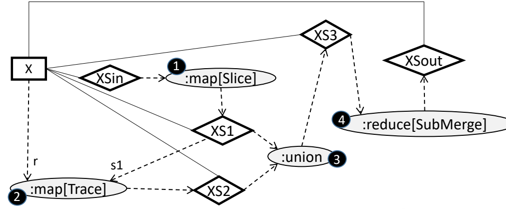


Figure 5.4: Illustration of the megamodel slice scenario.

Rule	Element under assessment	Dependant elements
CD1	Class	-Owned attributes and methods. -Associations connected to class. -Attributes/methods in other classes using types introduced in this class. -Subclasses.
SD1	Term (portion of an expression)	-Associated expression.
SD2	Expression (guard/action)	-Associated message.
SD3	Message	-Associated arrow (from source to target lifeline).
SD4	Arrow	-Arrows directly after the arrow in the sequence. -Message on the arrow.
SD5	Lifeline	-Arrows connected to the lifeline. -Messages on arrows connected to the lifeline.

Table 5.1: Dependency relations for CD and SD slicers.

is reached):

(1) Apply **map**[Slice](XSin) using the submodels in sub-megamodel XSin as the criterion to produce sub-megamodel XS1 containing the slice submodels.

(2) Apply **map**[Trace](XS1, X) to propagate the slice submodels in XS1 to corresponding submodels in neighbouring models to produce sub-megamodel XS2. Note that the r argument of Trace is taken from X while the s1 argument is taken from XS1.

(3) Apply **union**(XS1, XS2) to combine the megamodels from steps (1) and (2) to produce XS3.

(4) Apply **reduce**[SubMerge](XS3) to merge the multiple submodels of each model of X into a single submodel for each model. The result is a sub-megamodel XSout.

5.4 PSD Example

In this section, we demonstrate our slicing approach on the PSD example presented in Section 5.1.

5.4.1 Megamodels of class and sequence diagrams

For the purpose of the example presented here, we instantiate our general framework such that its input is a system megamodel X given by a class diagram CD, a sequence diagram SD, and a

relationship CD − SD between them. Note that, although these are both UML diagrams, we are treating them separately for the sake of this example. In general, not all models in a megamodel have to be UML diagrams.

Assume we are given some known change on the megamodel, which represents the slicing criterion $S_c[X]$ used as input to our algorithm. As stated in Section 5.2, we also assume that we are provided with correct class diagram and sequence diagram model slicers similar to those presented in [Lano and Rahimi(2010)] and [Noda *et al.*(2009)], respectively.

For simplicity, we define our own CD and SD slicers for this example as follows:

• CD slicer works with the dependency rule shown as CD1 in Table 5.1: If a class is being considered for impact assessment, then all of its attributes, methods, associations linked to it and its subclasses are considered dependant on it and could potentially be impacted. They are therefore to be added in the slice.

• SD slicer works with the dependency rules shown as SD1 − SD5 in Table 5.1: If a term, i.e., any portion of an expression (e.g., a guard or an action) in a message, is being considered for impact assessment, then its associated expression could be impacted. Similarly, if an expression (e.g., a guard or an action) is being considered, its associated message should be included in the slice. Other rules for impact assessment of messages, arrows and lifelines are shown in the table.

Note that both slicers satisfy Definition 17, i.e., they are correct and minimal. We also assume that the set of traceability relationships in CD − SD expresses all and only the dependencies between the CD and SD in our system megamodel.

5.4.2 Slicing of PSD megamodel

Recall the PSD megamodel presented in Figure 2.9 which we refer to as PSD. The models represented by PSD are in Figs. 2.2 and 2.3.

There are three threads running in parallel in the sequence diagram: the top thread describes the behaviour of the Redundant Switch; the middle thread describes the behaviour when the driver requests to open the door, and the bottom thread describes the behaviour when the driver requests to close the door. The relationship R : CD − SD is a unidirectional traceability relationship that goes from SD to CD, since the objects and terms of SD are dependent on classes, attributes and methods in CD. The traceability between the two models is given implicitly by the SD referencing parts of the CD.

As described in Section 5.1, let us consider a scenario where the system changes, and the redundancy is removed by deleting the Redundant Switch class from the CD. This change represents our slicing criterion given by the megamodel fragment with detail shown in Figure 5.5. Note that only the class itself is considered for the impact assessment and not its methods, attributes and associations linked to it.

We now demonstrate the application of the forward megamodel slice algorithm presented in Figure 5.1 on the megamodel PSD and the criterion megamodel fragment S_c[PSD].

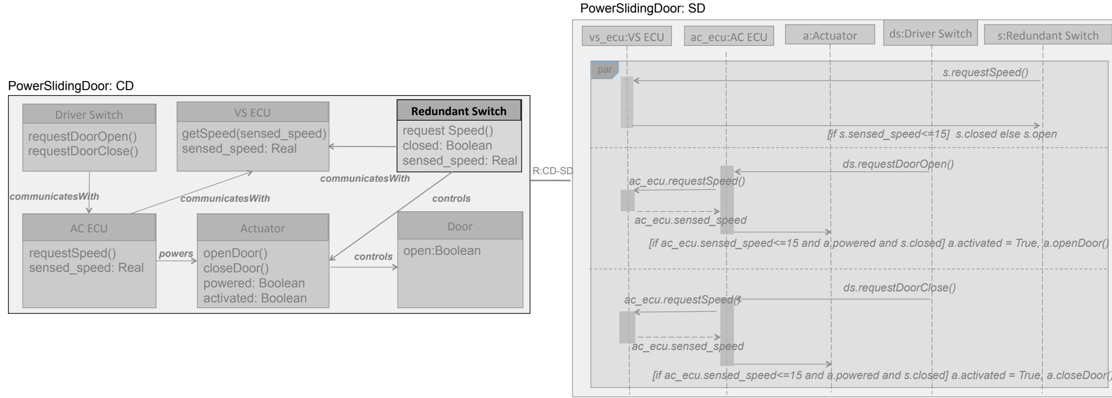


Figure 5.5: Slicing criterion S_c[PSD].

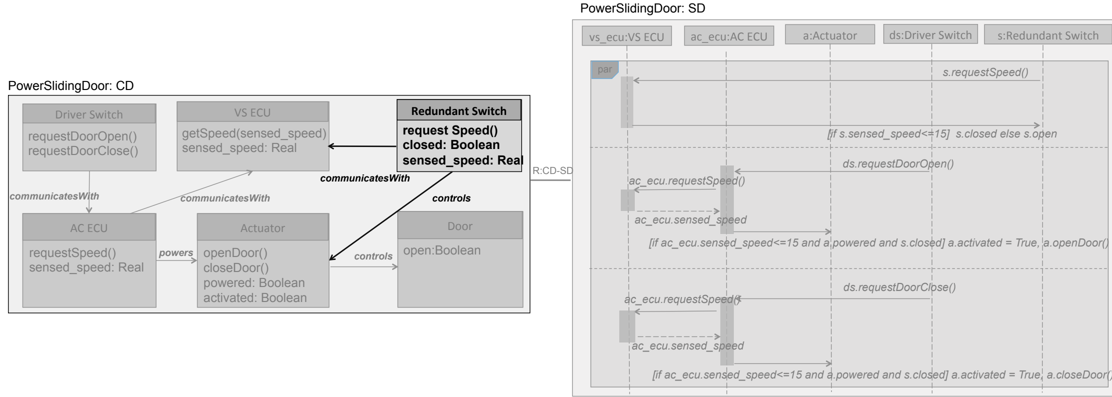


Figure 5.6: Result of level 1 slicing in 1st iteration.

Line 1 (Initialization): The current slice is initialized to the criterion S_c[PSD] shown as the highlighted parts of Figure 5.5.

1st iteration of the outer loop (lines 2-19):

Lines 4-9 (Expansion Level 1): The temporary result S_1[PSD] is initialized to the empty set. Then in lines 5-9, we iterate through the model fragments in the current slice shown in Figure 5.5. The CD is considered first and the CD slicer is used. Based on the dependency rule CD1 in Table 5.1, since the Redundant Switch class is being impacted, all of its attributes and methods are added to the slice and stored in S_1[PSD] on line 8. Since there are no other model fragments to consider on line 5, the loop exits with S_1[PSD] as shown by the highlighted parts in Figure 5.6.
Lines 10-17 (Expansion Level 2): Up to this point, $\mathsf{R : CD - SD}$ has not been considered in the slicing. In this expansion level, we do use it. First, the level 2 expansion temporary result S_2[PSD] is initialized on line 10 to the empty set. The outer iteration (lines 11-17) is over the model fragments from the level 1 expansion. We first consider the CD. On the opposite end of

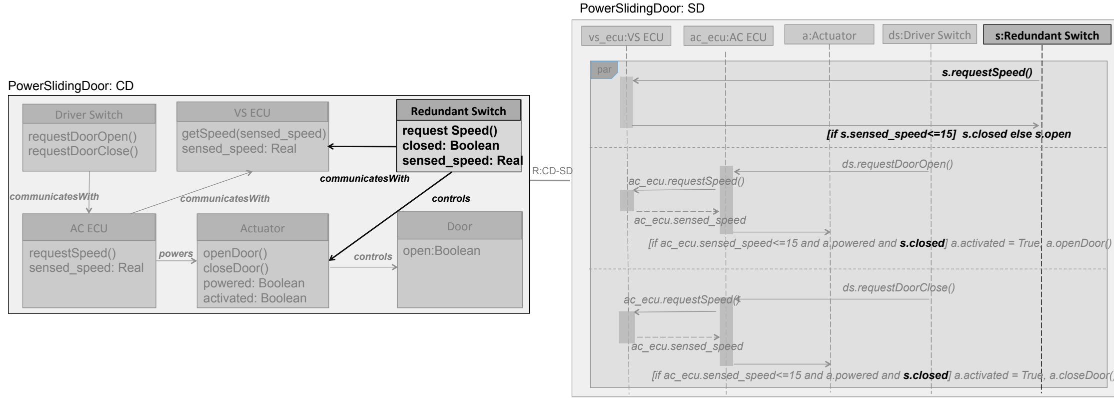


Figure 5.7: Result of the 1st iteration.

R : CD – SD is the PowerSlidingDoor : SD (which is M' in the algorithm on line 13). On line 14, we trace through R : CD – SD and add to $S_2[M']$ all the atoms related to those highlighted in the CD. This includes the Redundant Switch object and lifeline and all messages (or parts of them) that are traced back to attributes/methods of the Redundant Switch class in the CD. The result is added to S_2[PSD] on line 15 and can be seen in the highlighted parts of the SD in Figure 5.7. Since no other model fragments exist in S_1[PSD] on line 11, the loop exits.

Line 18: The combined result S[PSD] is computed by computing the union of the results of the level 1 and level 2 slices, and can be seen as the result of the 1st iteration of the algorithm in the highlighted parts of Figure 5.7.

Line 19: In this line, we check to see if any actual expansion has occurred. Since the condition is not met (i.e., the result of the 1st iteration did indeed expand on the initial criterion) we iterate one more time.

2nd iteration of the outer loop (lines 2-19):

The slicing criterion S[PSD] in this iteration is the result of the previous iteration shown in Figure 5.7. S_1[PSD] is reset again to the empty set.

Lines 4-9 (Expansion Level 1): First the CD is selected on line 5. Since none of the slicing dependency rules given in Table 5.1 apply, nothing is added to S_1[PSD] on line 8. Next, the SD is selected on line 5. Now, SD1 – SD3 rules for the SD slicer in Table 5.1 apply, and the SD slice is expanded to include the arrows of the top two messages and the entire expressions (and therefore messages and arrows) that the term s.closed appears in. This is seen in the highlighted parts of the SD portion of Figure 5.8.

Lines 10-17 (Expansion Level 2): In this level, tracing across the R : CD – SD relationship from the CD to the SD (recall this is a unidirectional traceability relationship), since no new elements

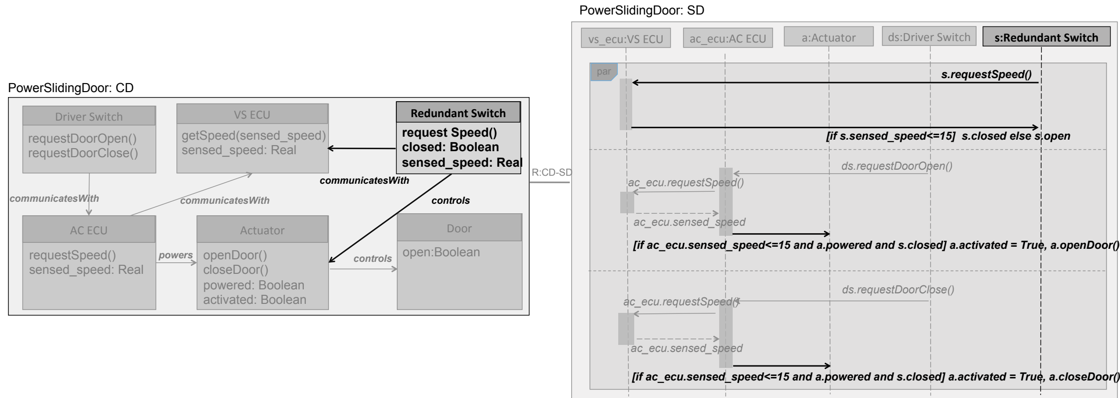


Figure 5.8: Result of 2nd iteration.

are introduced in the CD slice, nothing is traced to them in the SD. The result is an empty set.

Line 18: The results of the level 1 and the level 2 expansions are unioned and are reflected in the highlighted parts of Figure 5.8.

Line 19: Since an expansion (w.r.t. the initial slice for this iteration) has occurred, the condition does not hold, and we iterate one more time on the outer loop.

3rd iteration of the outer loop (lines 2-19):

In this iteration, neither the CD nor the SD are expanded in the first level expansion as none of the dependency rules for their respective slicers holds. Similarly, no new elements are added, and therefore going through the trace links does not identify any other elements to be added to the expansion in level 2. The condition on line 19 now holds (no expansion has occurred), and the main loop of the algorithm exits.

Line 20 (Return): The current slice, S[PSD], which is shown in the highlighted parts of Figure 5.8, is returned as the final result of the algorithm.

5.4.3 Post-processing

As suggested in Section 5.2, we perform a post-processing step, we expand the result of slicing algorithm shown in Figure 5.8 to ensure the model fragments are well-formed and contextual information for referential integrity is included.

For the CD, the VS ECU and Actuator classes are included since both endpoints of associations communicatesWith and controls are needed for well-formedness.

For the SD, the VS ECU, AC ECU and Actuator objects and their lifelines are included to satisfy the well-formedness constraint of arrows requiring their lifelines. Also, the execution bar on the leftmost lifeline is included, as both of its input and output arrows are included in the result of the slicing.

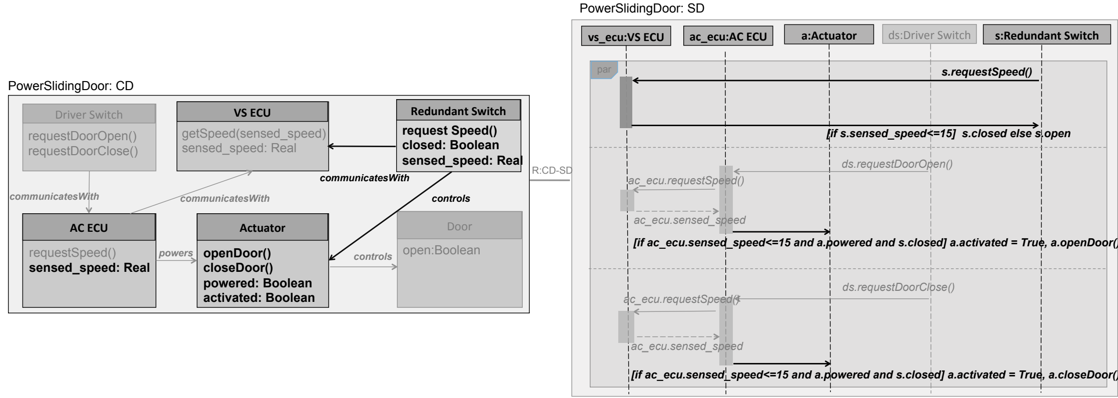


Figure 5.9: Output of algorithm after post-processing.

Finally, all the methods and attributes of the Actuator class, as well as the sensed_speed attribute of the AC ECU class and the AC ECU class itself are added to satisfy the referential integrity condition between the SD and the CD (they are all referenced in the SD).

The detail of the final megamodel fragment produced after the slicing and post-processing is shown in the highlighted parts of Figure 5.9. This can now be used to more efficiently complete the model evolution process by focusing only on the model parts impacted by the original deletion of Redundant Switch in the CD.

5.5 Related Work

We identify three main categories of related work: work on model evolution, work on megamodeling operators, and finally, work on model slicing. We describe them below.

Model evolution. A survey on supporting the evolution of UML models in model-driven software development is presented in [Khalil and Dingel(2013)]. The scenarios that cause a model to change are discussed; these form the basis for megamodel evolution in our approach. In [Paige *et al.*(2016)], the authors discuss some of the key problems of evolution in MDE, summarize the key state-of-the-art, and present some new challenges in research in this area. The problem of model evolution with respect to megamodels is stated as a “dependency heterogeneity” challenge. The authors express the need for a sound, precise theory of heterogeneous dependencies between MDE artefacts, as well as compliant and pragmatic tool support, both of which are complimentary to and/or are part of our current work.

Megamodeling operators. A formal approach to megamodeling, called *Mapping-Aware Megamodeling*, is presented in [Diskin *et al.*(2013)]. Our notion of a megamodel is consistent with it. The approach also describes category theory-based operations on the mapping-aware megamodels, but does not address megamodel slicing. In previous work [Salay *et al.*(2015)], we presented a set of

operators (Map, Filter, Reduce) that can be applied at the megamodel level. We are not aware of any other work in the area of applying operators at the megamodel level, and specifically, we have not seen any work addressing slicing of megamodels.

Model Slicing. We divide this area into work on *specific* model slicers, work on *generic* model slicers and work on slicing *multiple* models.

Specific Model Slicers. Numerous approaches have appeared in the literature describing slicers for specific model types. For example, [Kagdi *et al.*(2005)] defines context-free model slicing and presents an algorithm for computing slices on UML class models. [Lano and Rahimi(2010)] also considers UML models, namely, class diagrams, individual state machines, and communicating sets of state machines. The approach achieves slicing of these models using model transformations. An approach for slicing state-based models, in particular, EFSM (extended finite state machine) models, is discussed in [Korel *et al.*(2003)]. Finally, [Lallchandani and Mall(2011)] proposes a slicing technique for UML architectural models, and demonstrates the uses of slicing for different purposes such as regression testing and understanding large architectures. Many other approaches (e.g., [Noda *et al.*(2009)], [Lano and Rahimi(2010)]) are presented in the literature and can all be used as part of our framework as specific model type slicers for each of the model types in our heterogeneous megamodels.

Generic Model Slicers. Generic model slicing has also been studied in the MDE community. For example, the major contribution of [Blouin *et al.*(2011), Blouin *et al.*(2015)] is the Kompren language, which provides a generic approach to define a model slicer for any domain-specific metamodel. The approach permits developers to either use "strict slicers" that output models which conform to their expected metamodel, or to define "soft slicers" that can output nonconforming models or even outputs that are not models. Although Kompren can be used for identifying specific type slicers in our framework, it is not applicable for megamodel slicing, where a megamodel slicer has to carefully invoke the specific type slicers. The work in [Clark(2011)] defines slicing at a theoretical level, whereas we focus on a more pragmatic approach. Also, the same work focuses on dynamic slicing, as does the transformation slicing work in [Ujhelyi *et al.*(2011)], whereas our approach is considered a static slicing approach. As far as we know, none of the approaches in this category directly address megamodel slicing (whether the megamodels are heterogeneous or not).

Slicing Multiple Models. Although the work presented in [Clark(2011)] does not primarily focus on megamodel slicing, it briefly discusses heterogeneous slicing as the union of individual slicers. A slicing theory is presented at a high level and does not go into the details of implementing a megamodel slicing algorithm. From the modeling and safety community, [Nejati *et al.*(2012)] proposes a batch model slicer for slicing SysML models related to safety requirements. [Falessi *et al.*(2011)] presents a prototype tool called SafeSlice which performs the slicing needed in [Nejati *et al.*(2012)]. This line of work performs slicing on specific model types, whereas our work is a generic slicing approach. Also, the presented approach is amorphous slicing, where the result of the slice is not a model fragment of the original system. For example, transitions are added to sliced state-machines

in order to preserve their behaviour. Our current approach only considers slices to be fragments of the original model (non-amorphous); however, we do plan to look at amorphous slicing in future work.

5.6 Chapter Summary

In this chapter, we presented a general algorithm for slicing of heterogeneous model collections represented using megamodels and illustrated the algorithm on an automotive example. We analyzed the algorithm and showed that it behaves as expected with respect to termination, correctness, time complexity and minimality. We also demonstrated a version of the algorithm using the **map** and **reduce** operators from Chapter 4. We discussed the issues concerning slice well-formedness and referential integrity as well as how to generalize the algorithm to support arbitrary relationship types, N-ary relationships and nested megamodels.

Part III

Assurance Case Management

Chapter 6

Background: Assurance

In this chapter, we present a number of core concepts required for describing our contributions in the rest of this thesis. Section 6.1 motivates our focus on the automotive domain and Section 6.2 presents material on the ISO 26262 standard. Section 6.3 presents material on assurance cases and notations. Finally, Section 6.4 details our survey results on assurance case tools over the past 20 years, motivating the need for sound and effective assurance case management tools, especially for assurance case maintenance, which is one of the contributions of this thesis. The survey was published in [Maksimov *et al.*(2018)], where my contributions focused on providing feedback on the assurance case tools in the literature, guiding their assessment and presentation of the survey results.

6.1 Software Development in the Automotive Domain

This thesis is part of a larger collaborative research project with an automotive OEM. The main goal is to improve the safety, security and dependability of advanced automotive software systems, by developing methods and tools for creating and managing effective and practical assurance cases in the automotive industry. The project also aims to provide methods that support system and safety evolution correctly, quickly (via scalable automated tool support) and while facilitating product-level and product-line reuse. One specific subgoal is to develop methods and tools for impact analysis of assurance cases - which is one of the main contributions of this thesis.

To motivate the need for this work in the automotive domain, it is worth understanding a few things about software development in this domain and what makes it different from other domains.

Figure 6.1[1] shows how many MLOC (Million Lines Of Code) an average product from each domain contains. Car software (at the bottom of the chart) requires approximately 100 MLOC, much more than an F-35 fighter jet for example (at the top of the chart).

Moreover, the automotive domain has a lot more to do compared to other domains (see complexity in lines of code in Figure 6.2) in a lot less time (see development time in Figure 6.3).

[1] Source: Information is Beautiful (https://informationisbeautiful.net/visualizations/million-lines-of-code/)

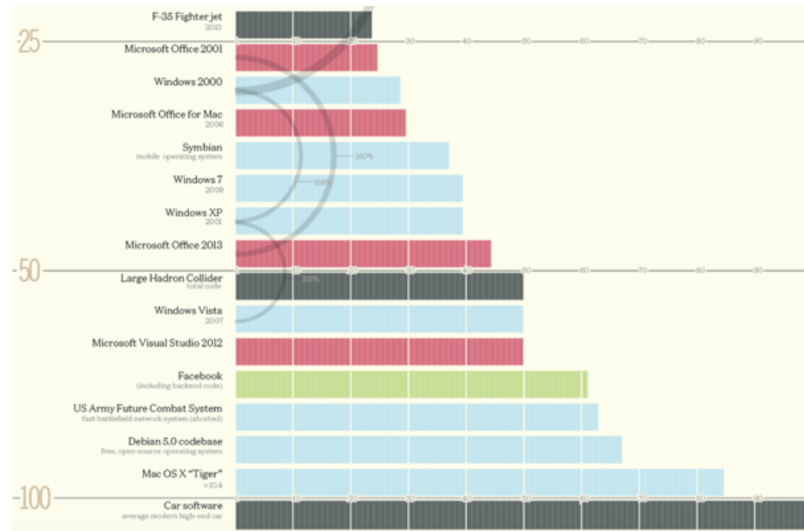


Figure 6.1: MLOC by product.

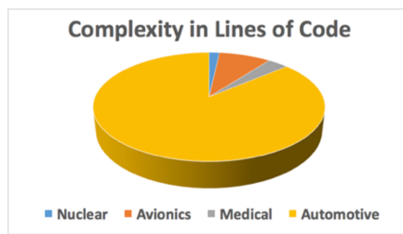


Figure 6.2: Complexity in lines of code.

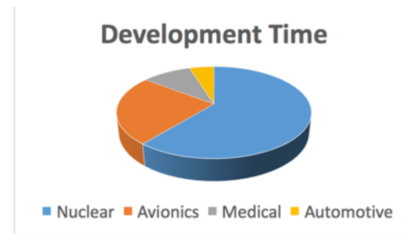


Figure 6.3: Development time.

Given this challenge of complexity and time crunch, an obvious solution is to use an incremental development approach. The idea of such an approach is to reuse existing safety assurance arguments for minor design changes. However, this has been shown not to work in practice, as it has led to an increasing number of recalls in the industry as can be seen in Figure 6.4[2].

This has motivated the need for *sound* and *efficient* approaches for incremental certification, which ensure the safety of products being developed in an incremental development fashion.

6.2 The ISO 26262 Standard

ISO 26262 is a standard that regulates functional safety of road vehicles. It deals with the electrical and electronic components of automotive vehicles - including software. The standard recommends conducting a Hazard Analysis to identify and categorize hazardous events in the system and its

[2] Source: Software Is Eating the Auto Industry (https://www.strategyanalytics.com/strategy-analytics/blogs/automotive/infotainment-telematics/infotainment-telematics/2017/08/25/software-is-eating-the-auto-industry)

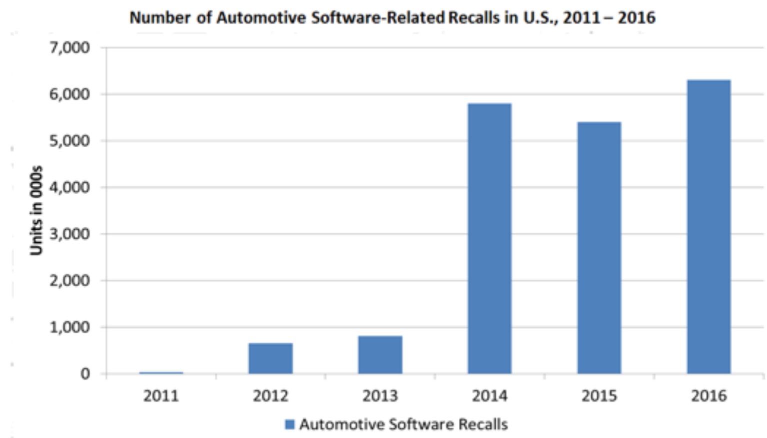


Figure 6.4: Automotive Software Related Recalls 2011-2016.

environment, and to specify safety goals and integrity levels related to the mitigation of the associated hazards. The standard has 10 parts as seen in Figure 6.5, and covers planning, development, maintenance, operation, and decommissioning of software and electronics in automotive vehicles. In this work, we focus on one aspect, "Product Development at the Software Level" (Part 6), and refer to Part 9 which explains Automotive Safety Integrity Levels (ASILs).

Since ISO 26262 is complex, existing commercial tools help companies comply with the standard. Some of these tools include Application Lifecycle Management (ALM)[3] and compliance tools such as Intland's codeBeamer[4] and Medini Analyze[5]. These tools mainly offer a process and document driven system and safety assurance approach, and help ensure process workflows are followed. Some of these tools provide traceability and some automation, but significant manual work is required using Excel spreadsheets and Word documents. Such tools have little support for incremental safety analysis and do not offer an explicit model of the product safety case.

6.2.1 ASIL Allocation and Propagation

An ASIL refers to an abstract classification of inherent safety risk in an automotive system or elements of such a system. ASIL classifications are used within ISO 26262 to express the level of risk reduction required to mitigate a specific hazard, with ASIL D representing the highest and ASIL A the lowest. If an element is assigned QM (Quality Management), it does not require safety management. The ASIL assessed for a given hazard is then assigned to the safety goal set to address that hazard and is then inherited by the safety requirements derived from that goal following ASIL propagation rules. The higher the ASIL, the more rigorous the application of ISO 26262 has to be, i.e., the more requirements need to be fulfilled.

[3]http://info.perforce.com
[4]https://codebeamer.com/
[5]https://www.ansys.com/products/systems/ansys-medini-analyze

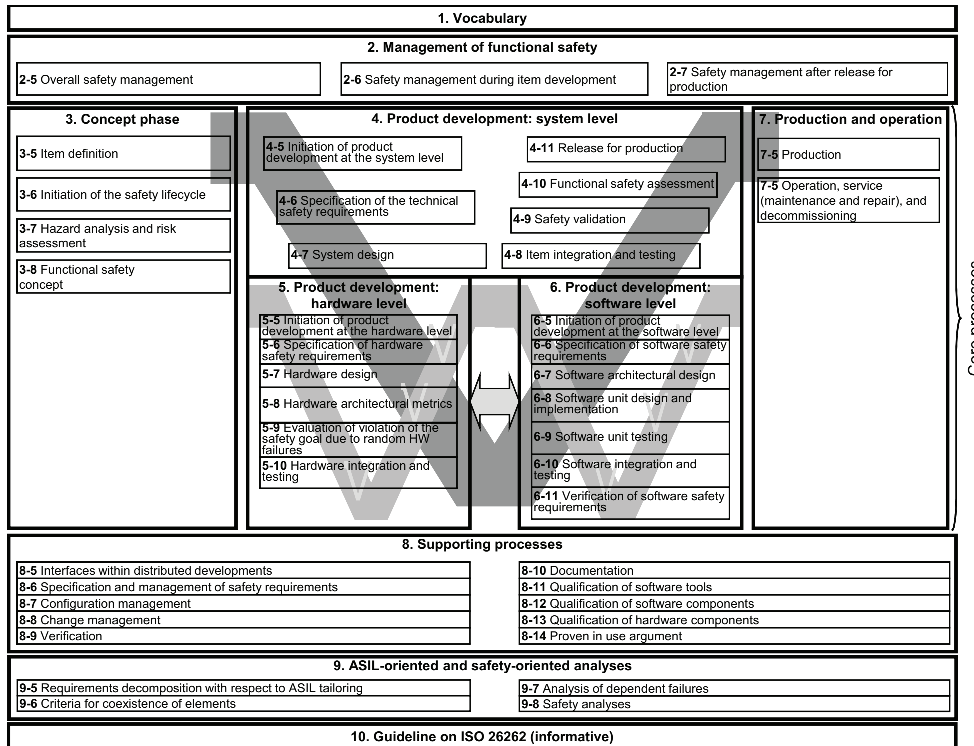


Figure 6.5: The 10 parts of ISO 26262.

6.2.2 ASIL Decomposition

The method of ASIL tailoring during the design process is called "ASIL decomposition". When allocating ASILs, benefit can be obtained from architectural decisions, including the existence of sufficiently independent architectural elements (as in the redundancy in the original PSD system). This offers the opportunity to implement safety requirements redundantly by these independent architectural elements, and to assign a potentially lower ASIL to these decomposed safety requirements as seen in Figure 6.6 (refer to Figure 2 in Part 9 of the standard for ASIL decomposition schemes.).

Furthermore, ISO 26262 requires the production of over 100 work products, achieved via various requirements and methods used in the different phases of software development. For example, Section 9 of Part 6 of ISO 26262 discusses Software Unit Testing, and Section 9.5 outlines the required work products for it. One of these work products is 9.5.1: Software Verification Plan which results from requirements 9.4.2-9.4.6 in the same section. Consider one of these requirements, 9.4.3, which describes which software testing methods can be used. These methods clearly link to ASILs. Specifically, Figure 6.7, lists various methods for software unit testing and how they relate to the four ASILs. The degree of recommendation to use the corresponding method depends on the ASIL and is categorized as follows: "++" indicates that the method is highly recommended for the identified ASIL (we interpret this as "required"), "+" – that the method is recommended for the identified

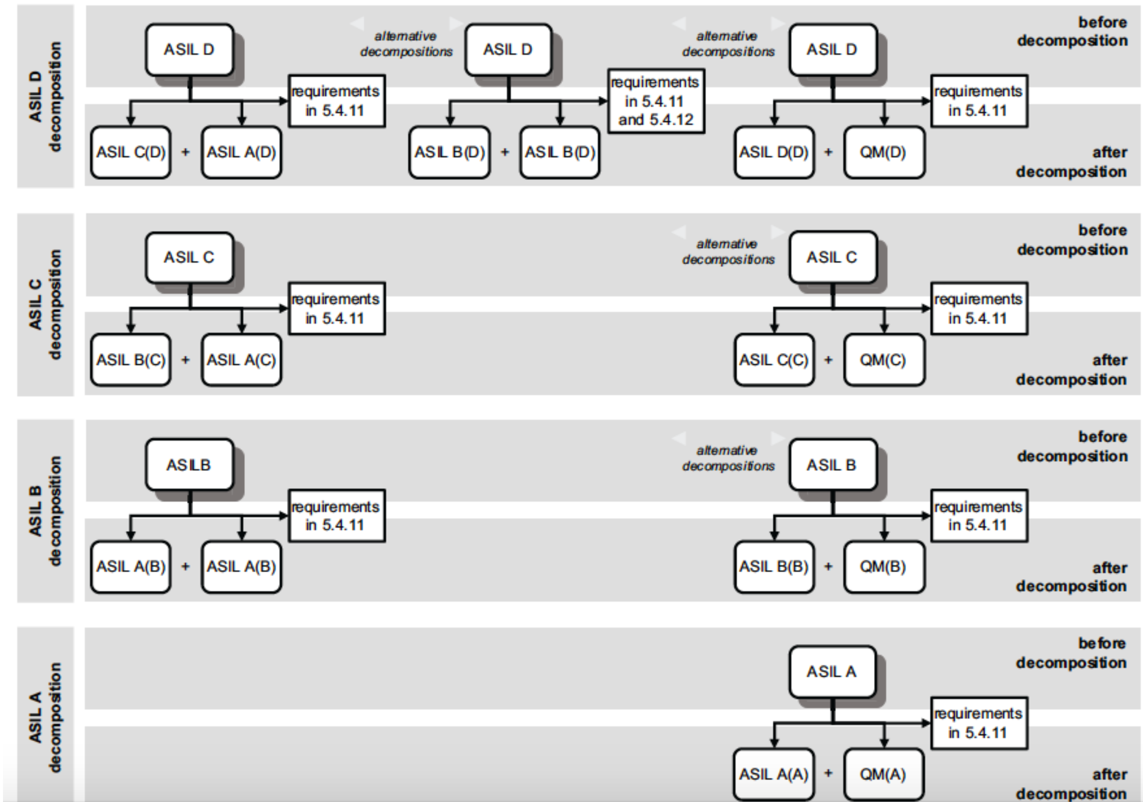


Figure 6.6: ASIL decomposition schemes from ISO 26262.

ASIL, and “o” – that the method has no recommendation for or against its usage for the identified ASIL. For example, methods 1a, 1b, 1e in Figure 6.7 are required for unit testing for ASIL C. An increased ASIL D, now requires methods 1c and 1d which were only recommended for ASIL C.

6.2.3 Goal Refinement in ISO 26262

An important element to be accounted for in ISO 26262 is the relationship existing between so-called *safety goals* (SGs), *functional safety requirements* (FSRs), *technical safety requirements* (TSRs), *hardware safety requirements* (HWSRs), and *software safety requirements* (SWSRs).

In brief, an SG is a top-level safety requirement that is in place to prevent or mitigate some associated hazards so as to avoid unreasonable risk. FSRs, TSRs, HWSRs, and SWSRs are also safety requirements, derived from SGs, but expressed at different levels of description of the system design. FSRs may be thought of as a technology independent safety requirements derived from SGs, TSRs are technology dependent safety requirements derived from implementation of FSRs, SWSRs and HWSRs are specific safety requirements implemented as part of the software and hardware design. This relationship between SGs, FSRs, TSRs, HWSRs, and SWSRs can be thought of as

	Methods	ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

Figure 6.7: Methods for software unit testing - ISO 26262 Part 6.

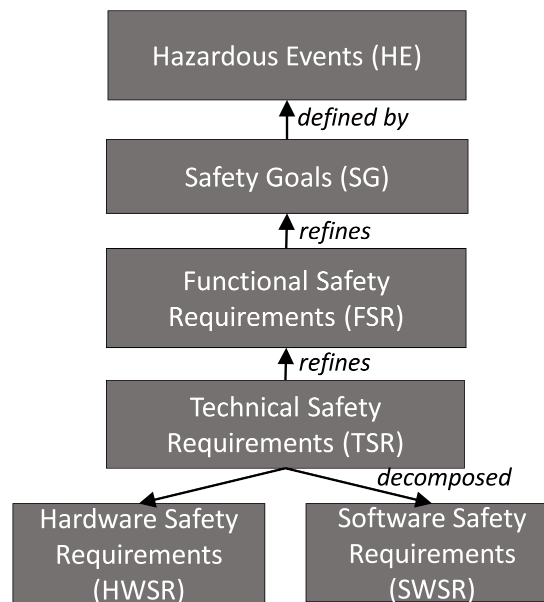


Figure 6.8: Goal refinement in ISO 26262.

refinement as follows: SGs are stated, FSRs refine SGs, TSRs refine FSRs, and so on (see Figure 6.8). If viewed in this sense, the relationship between SGs, FSRs, TSRs, HWSRs, and SWSRs may be represented as the tree [Dardenne *et al.*(1993)] shown in Figure 6.8. Such a tree makes an assurance case that all safety requirements have been satisfied.

6.3 Assurance Cases

In regulated safety-critical domains, such as the aerospace and nuclear domains, certification bodies often require systems to undergo a stringent safety assessment procedure to show their compliance to one or more safety standards. Quality standards mandate the creation of quality-specific requirements and assurance cases. For example, ISO 26262 describes how safety requirements, levels of specification and a safety case for these must be produced to certify the safety of a vehicle.

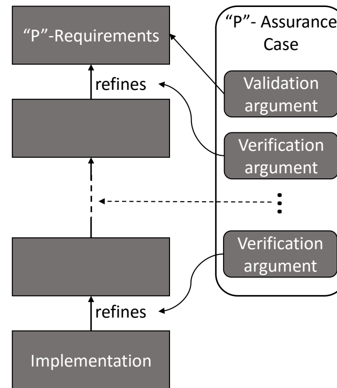


Figure 6.9: Software development with assurance cases. "P" represents the quality being assured.

Figure 6.9 shows a simplified view of software development when assurance is considered. Here, "P" represents the quality of interest to be assured, e.g., safety, privacy, security, etc. For the given quality, the system requirements are determined and traced to the implementation through a series of specification levels that can include both requirements and design refinements. An assurance case for such a process must contain a validation argument for the initial requirements as well as verification arguments for each refinement step.

6.3.1 Modeling Assurance Cases

An *assurance case* is an artifact that shows how important claims about the system (e.g., requirement satisfaction) can be argued for, ultimately from evidence obtained about the system such as test results, expert opinion, etc. Several approaches to modeling assurances cases have been proposed, including GSN [Kelly and Weaver(2004)], CAE [Bloomfield and Bishop(2010)], KAOS-based [Brunel and Cazin(2012)] and, more recently, SACM [de la Vara(2014)]. All of these approaches agree that an assurance case must contain three core concepts: claims, arguments and evidence. In order to develop a generic model management framework for assurance case reuse, we use the abstract metamodel shown in Figure 6.10 for an assurance case based on these core concepts rather than choosing a particular concrete approach from the literature.

A Claim represents a statement about the system or some part of it. The state attribute represents the truth state (e.g., true, affirmed, refuted, etc.) of this statement. An Evidence element represents some set of data obtained about the system. These could include test results, analysis results, an expert opinion, a formal correctness proof, etc. Here, the state attribute indicates the validity state of the evidence - e.g., currently valid, is stale and must be regenerated, etc. The Argument elements connect claims to each other and to evidence. An argument takes zero or more claims and evidence as input and has one claim as a conclusion. Semantically, it represents how the conclusion follows from the input claims and evidence.

There is natural derived dependency relation connecting atoms of an assurance case.

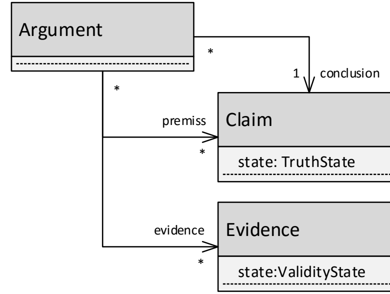


Figure 6.10: Generic assurance case metamodel AC.

Definition 22 (Assurance case dependency relation) *Given an assurance case $A : AC$ defined according to the metamodel in Figure 6.10, the* dependency relation $ACdep \subseteq atoms_A \times atoms_A$ *for all atoms $x, x', x'' \in A$ is defined as follows:*

- *(reflexive) $ACdep(x, x)$;*
- *if x is a* Claim *that is the* conclusion *of* Argument *x' then $ACdep(x, x')$;*
- *if x is an* Argument *that has a* premise *or* Evidence *x' then $ACdep(x, x')$;*
- *(transitive) if $ACdep(x, x')$ and $ACdep(x', x'')$ then $ACdep(x, x'')$.*

Furthermore, we make the following semantic assumptions about assurance cases:

- If assurance case $A : AC$ is considered to be complete and correct then the truth state of claim c can only be affected by the truth state of some claim x or by the content of the evidence x iff $ACdep(c, x)$.
- If the truth state of input claims and the content of evidence for an argument do not change then the truth state of the conclusion claim cannot change.

We now present some commonly used graphical assurance case notations: the Goal Structuring Notation (GSN), Claims, Arguments and Evidence (CAE), and OMG's recent Structured Assurance Case Metamodel (SACM).

6.3.2 The Goal Structuring Notation (GSN)

The most commonly used representation for safety cases is the graphical Goal Structuring Notation (GSN) [GSN(2011)], which is intended to support the assurance of critical properties of systems (including safety). GSN is comprised of six core elements as summarized in Figure 6.11.

Arguments in GSN are typically organized into a tree of the core elements shown in Figure 6.11. The root is the overall goal to be satisfied by the system, and it is gradually decomposed (possibly with strategies) into sub-goals and finally into solutions, which are the leaves of the safety case.

Connections between goals, strategies and solutions represent *supported-by* relations, which indicate inferential or evidential relationships between elements. Goals and strategies may also, but not necessarily, be associated with some contexts, assumptions and/or justifications by means of *in-context-of* relations, which declare a contextual relationship between the connected elements.

For example, consider the safety case presented in Figure 6.12. The overall goal (G1) is that the "Control System is acceptably safe to operate" given its role, context and definition, and it is decomposed into two sub-goals; (G2) for eliminating and mitigating all identified hazards and (G3) for ensuring the system software is developed to an appropriate safety integrity level (SIL). Assuming that all hazards have been identified, G2 can in turn be decomposed into three sub-goals by considering each hazard separately (S1), and each separate hazard is shown to be satisfied using evidence from formal verification (Sn1) or fault tree analysis (Sn2). Similarly, under some specific context and justification, G3 can be decomposed into two sub-goals, each of which is shown to be satisfied by the associated evidence.

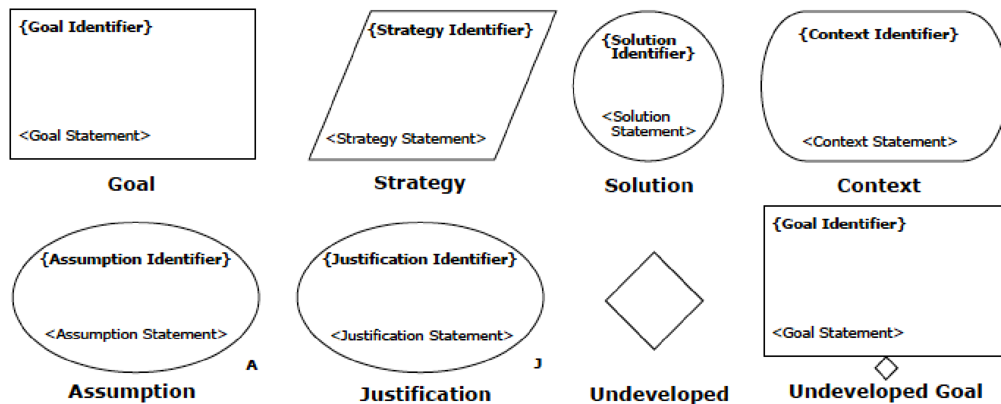


Figure 6.11: Core GSN elements from [GSN(2011)].

In addition to the elements described in Figure 6.11, there is a GSN extension, introduced to help with the creation of argument patterns. It adds a set of elements to assist with abstraction modelling. There is also a modular extension to GSN. The interested reader can find the details for both of these features in [GSN(2011)].

6.3.3 Claims, Arguments and Evidence (CAE)

CAE (Claims, Arguments and Evidence) was developed in the U.K. by Adelard. It is a graphical notation, and in Figure 6.13, we provide a table with definitions for its elements, taken from [Adelard(2018)].

CAE is similar to GSN in providing a tree structure with a fundamental claim at the top and evidence at the leaves of the tree. It also suggests the inclusion of contextual information with the claim. Adelard provides a software tool, ASCE (Assurance and Safety Case Environment) for

the creation of assurance cases using this notation. Figure 6.14 [Adelard(2018)] demonstrates the graphical representation of CAE elements and how they can be linked together.

The CAE notation is compatible with the Structured Assurance Case Metamodel (SACM), which we describe next.

6.3.4 Structured Assurance Case Metamodel (SACM)

The Structured Assurance Case Metamodel (SACM) [OMG(2015)] is standardized by the Object Management Group (OMG). The goal of the metamodel is to promote a model-based approach in the process of System Assurance, which is currently a manual approach which produces assurance cases that are typically not machine-consumable. SACM is created to support structural argumentation approaches such as GSN and CAE. SACM captures not only fundamental concepts in the process of System Assurance such as Claims and the relationships between them, but also concepts such as Artifacts and Terminologies. This allows supporting evidence and information involved in the argument to be modelled in greater precision. In addition, SACM promotes modularity; assurance cases are organized in packages, which in turn organize argumentations, evidence and terminologies in corresponding packages. Finally, SACM also allows external information (such as external models and/or documents) to be linked via the facilities provided.

SACM, however, has some limitations. For example, it does not provide an ISO26262-specific assurance case template, and does not provide a way of performing incremental assurance using model management or address product lines.

Overall, SACM is organized in five packages: the *Base* package provides the foundation of SACM, the *Argumentation* package captures the concepts used in arguing system properties (such as safety and/or security), the *Terminology* package captures the concepts used in expressing the arguments regarding system properties, the *Artifact* package captures the concepts used in providing evidence for the arguments made for system properties, and the *AssuranceCase* package captures the concepts in System Assurance, which combines all the elements in other SACM packages to form a System Assurance Case.

Both GSN and CAE can be mapped onto SACM argumentation aspects. The argumentation model thus provides the ability to exchange structured argumentation information created with a tool using either of these notations. The document available at [The GSN Working Group(2015)] provides examples of GSN implemented using SACM.

6.4 A Survey of Assurance Case Tools

Assurance cases (ACs) can be very complex; e.g., an assurance case for an air traffic control system may comprise over 500 pages and 400 referenced documents [Lewis(2009)]. Tools to support safety engineers in creating, maintaining and analysing ACs have been developed. For example, Resolute [Gacek *et al.*(2014)] can automatically generate ACs based on a system's architectural models,

while AGSN [Luo *et al.*(2017)] supports the assessment of an AC's validity. The development of these tools has been enabled by the introduction of formal syntaxes for ACs, such as the Goal Structuring Notation (GSN) [Kelly(1998)]. In this section, we aim to perform a systematic review of the progress made in the development of tools for ACs. To the best of our knowledge, this is the first such study. More specifically, the main contributions of this section are (1) a comprehensive list of AC tools developed over the past 20 years; and (2) an analysis of these tools according to their functionality.

The remainder of this section is organised as follows. First, we present a methodology for finding and comparing AC tools. Then, we present and summarise the findings and potential threats to validity. We conclude by discussing the implications of this survey.

6.4.1 Methodology

We carried out a Systematic Literature Review (SLR) in order to establish a complete list of AC tools and provide a comprehensive assessment of their features. Our SLR followed a simplified version of the guidelines proposed by [A. Kitchenham(2007)], as well as the search strategy proposed by [Zhang *et al.*(2011)]. It consists of three stages: **(1)** establishing the *quasi-gold standard* (QGS) through a manual search of different publication venues, **(2)** an automated literature search of digital libraries, e.g., Springer Link and IEEE Xplore, and **(3)** a web-based search for commercial tools and tools that may not have been mentioned in publications. We describe these steps below.

Manual Search and Establishing the QGS. A QGS is a set of high quality studies from the related publication venues on a research topic, e.g., domain-specific conferences and journals recognized by the community in the subject, for a given time span [Zhang *et al.*(2011)]. To create a QGS, relevant publication venues are identified and manually searched in order to retrieve studies that serve as a benchmark for the subsequent automated search. Through consultation with domain experts, we identified six major conferences and journals that published research on ACs: **(1)** SAFECOMP (International Conference on Computer Safety, Reliability, & Security), **(2)** HASE (International Symposium on High Assurance Systems Engineering), **(3)** IMBSA (International Symposium on Model-Based Safety and Assessment), **(4)** ISSRE (International Symposium on Software Reliability Engineering), **(5)** Reliability Engineering & System Safety (journal), and **(6)** COMPSAC (International Conference on Computers, Software & Applications). We performed a manual search through the proceedings of these venues including all associated workshops, for 2015-17 inclusive, yielding 10 relevant AC tool papers which established our QGS.

Defining the Search String and Performing the Automated Search. A careful examination of the papers in our QGS constructed the search string to be *"("Safety Assurance" OR GSN OR SACM OR "Safety Case" OR "Safety Cases" OR "Assurance Case" OR "Assurance Cases" OR "Safety Compliance") AND (Editor OR Tool OR Editors OR Tools OR Toolset OR Toolsets)"*. We used it to conduct an automated literature search on IEEE Xplore, Engineering Village, ACM

Digital Library and Spring Link[6], combined with the criterion that the papers were in English and published after 1998.

IEEE Xplore, Engineering Village, ACM Digital Library, and Springer Link returned 112, 739, 21, and 80 papers respectively, for a total of 952 papers. We checked the resulting papers against our QGS which captured 8/10 papers, achieving the recommended 80% sensitivity [Zhang *et al.*(2011)]. After filtering out duplicate papers, papers not accessible in full text, irrelevant papers (based on a manual review of their abstracts or the full text), we identified 82 papers.

Performing the Web-Based Search. To obtain knowledge about commercial AC tools, tools that were published but were not found by our literature search, or tools that simply were not mentioned in publications, we conducted a web-based search[7] using Google as the search engine. We used the same search string as for the literature search and viewed the first 100 results. This step yielded eight additional tools.

Evaluating the Tools. Having read all of the publications and the resources gathered by our searches, we established six distinct recurring tool functionalities, using them as the basis for our evaluation. These functionalities are categorized as AC creation, maintenance, assessment, collaboration, reporting and integration (see Table 6.1). We then defined four levels of tool support for each of the categories, ranging from D (no support) to A (strong support), thus creating our grading criteria. We then graded each tool's degree of support for each category, using information from the publications and the web resources. Since information in some of the publications can be out of date, we made an effort to use the newest publications so as to arrive at a more accurate evaluation. Please note that our evaluation is based purely on the information found in the above resources rather than on the hands-on testing of the tools.

6.4.2 Results

Our systematic literature review discovered a total of 46 AC tools. Eight of these tools (AssureNote [AssureNote(2018)], PREEVision [PREEVision(2018)], SMS Pro [Pro(2018)], Artisan GSN modeler [University of York(2018)], Assure-It [Shida *et al.*(2013)], SEAS [Ankrum and Kromholz(2005)], TurboAC [TurboAC(2018)] and eDependabilityCase [Lautieri *et al.*(2004)]) were discovered by our web search; two (MMINT-A [Fung *et al.*(2018)] and Resolute [Gacek *et al.*(2014)]) were identified with the help of domain experts, and the remainder were found by our literature search. Nine tools (AssureNote [AssureNote(2018)], DECOS Test Bench [Althammer *et al.*(2009)], e-Safety Case [Larrucea *et al.*(2017)], GSN CaseMaker ERA [Larrucea *et al.*(2017)], ISIS High Integrity Solutions [Larrucea *et al.*(2017)], PREEVision [PREEVision(2018)], SCAPT [Allan *et al.*(1998)], SEAS [Ankrum and Kromholz(2005)] and SMS Pro [Pro(2018)]) did not provide sufficient information allowing us to conduct an educated

[6] The literature search was conducted in the dates between 02.02.2018 - 19.02.2018.
[7] carried out on 25.02.2018

Feature Category	**Level of Tool Support**			
	D (No Support)	**C (Minimal Support)**	**B (Moderate Support)**	**A (Strong Support)**
Support for AC creation (Creation).	None	Basic support for the manual creation of ACs.	Partial automation or re-use in creating ACs is available (e.g. argument patterns and templates).	Automatic creation of complete ACs.
Support for maintaining ACs as they evolve (Maintenance).	None	Manual editing with no guidance on affected parts provided.	Tracking of relevant artefacts (e.g. system models and evidence), notifying user of changes and/or indicating their potential impact on the AC.	Automatic updates of ACs to reflect changes in the relevant artefacts (e.g., evidence, system models, requirements specifications).
Support for assessing ACs (Assessment).	None	Support for manual annotations to indicate potential problems.	Support for syntactical checks (e.g., for well-formedness, completeness and/or consistency).	Syntactic and semantic checking (e.g., validity of overall argument given its supporting arguments and evidence).
Support for collaboration between users (Collaboration).	None	A basic concurrent multi-user environment.	Additional features such as user access/permission management.	A complex multi-user environment (e.g., change requests and change reviews).
Support for creating reports from ACs (e.g. for certification purposes or for different stakeholders) (Reporting).	None	Generic reports with no user configurability, limited range of document formats and/or limited content.	Some user configurability, in multiple document formats and/or containing more content.	High user configurability, extensive document formats and/or detailed/interactive content (e.g., generating different reports).
Support for other design/assurance lifecycle processes (e.g. RE specs, hazop, verification) (Integration).	None	Manual integration.	Some support (e.g., bundling with specific third-party tools).	Extensive support for many other design/assurance lifecycle processes.

Table 6.1: Tool functionality categories and the corresponding degrees of support.

evaluation, and are thus excluded from further discussion[8].

Out of the 37 AC tools (see Table 6.2), 32 offer support for GSN [GSN(2011)]. Some exceptions to this are Modus [Sabetzadeh *et al.*(2013)] (a plug-in for Enterprise Architect), ACBuilder [Kawakami *et al.*(2016)], NOR-STA [Gorski *et al.*(2012)], etc., which have their own notations. Multiple tools (e.g., CertWare [Barry(2011)] and ASCE [Netkachova *et al.*(2015)]) also offer support for a variety of different notations, such as the Structured Assurance Case Metamodel (SACM) [OMG(2015)] and the Claims-Arguments-Evidence (CAE) [Bloomfield and Bishop(2010)] notations, in addition to others. Our findings also show that most of the tools are not domain specific, meaning that they can be used to construct ACs for military, automotive, medical, and nuclear systems, among others. Exceptions to this are tools such as ACBuilder [Kawakami *et al.*(2016)] (hardware security analysis) and TurboAC [TurboAC(2018)] (medical devices). Non domain specific tools (e.g., D-Case Editor [Matsuno *et al.*(2010)]) have been marked with a hyphen under the domain column in Table 6.2.

[8] A table listing more information about each evaluated tool, such as where it was produced, how it was discovered, a link to the tool, its availability, its supported notations and domain, can be accessed at goo.gl/A4yWs9.

Tool name	Supported notations	Domain
ACBuilder [Kawakami *et al.*(2016)]	Textual	Hardware security analysis
ACCESS [Larrucea *et al.*(2017)]	GSN	-
ACEdit [Larrucea *et al.*(2017)] https://github.com/arapost/acedit	GSN, ARM	-
AdvoCATE [Denney and Pai(2017)] https://ti.arc.nasa.gov/tech/rse/research/advocate/	GSN, SACM, Bowtie	-
AGSN [Luo *et al.*(2017)] https://github.com/AGSNeditor/development	GSN	-
ASCE [Netkachova *et al.*(2015)] https://www.adelard.com/asce/choosing-asce/index/	CAE, SACM, GSN, Bowtie	-
Assure-It [Shida *et al.*(2013)]	GSN	-
Astah GSN [Larrucea *et al.*(2017)] http://astah.net/download	GSN, ARM, SACM	-
Artisan GSN modeler [University of York(2018)]	GSN	-
AutoFOCUS3 [Cârlan *et al.*(2017)] https://af3.fortiss.org/download/	GSN	Distributed, reactive, embedded software systems
CertWare [Barry(2011)] https://nasa.github.io/CertWare/	ARM, CAE, GSN, EUROCONTROL	-
D-Case Communicator [Matsuno(2017)] https://mlab.ce.cst.nihon-u.ac.jp/dcase/login.html	GSN	-
D-Case Editor [Matsuno *et al.*(2010)] http://www.jst.go.jp/crest/crest-os/osddeos/en/tech.html	GSN, SACM	-
D-Case Weaver [Fujita *et al.*(2012)] http://www.jst.go.jp/crest/crest-os/osddeos/en/tech.html	GSN	-
D-MILS [Cimatti *et al.*(2015)] https://github.com/phy3rdh/DmilsMBAC	GSN	-
Eclipse & Papyrus extension [Huhn and Zechner(2009)]	GSN	-
eDependabilityCase [Lautieri *et al.*(2004)]	GSN	-
ENTRUST [Calinescu *et al.*(2017)] https://github.com/gerasimou/ENTRUST	GSN	Self-adaptive software
eSafetyCase Toolkit [Newton and Vickers(2007)]	GSN	-
ETB (Evidential Tool Bus) [Cruanes *et al.*(2013)] https://github.com/SRI-CSL/ETB	Claims table	-
Event-B extension [Laibinis *et al.*(2015)]	GSN	-
EviCA [Nair *et al.*(2015a)] Can be acquired by emailing the authors	GSN	-
GAGE [Bjornander *et al.*(2012)]	GSN	-
HiP-HOPS extension [Retouniotis *et al.*(2017)] http://www.hip-hops.eu/	GSN	-
ISCaDE [Larrucea *et al.*(2017)] http://www.iscade.co.uk/	GSN, ASCAD, WeFA	-
MMINT-A [Fung *et al.*(2018)] https://github.com/adisandro/MMINT	GSN	-
Modus [Sabetzadeh *et al.*(2013)] http://modelme.simula.no/Modus	KAOS	-
NOR-STA [Gorski *et al.*(2012)] https://www.argevide.com/purchase/assurance-case/	TRUST-IT Argument Representation	-
OpenCERT [Larrucea(2016)] https://www.polarsys.org/proposals/opencert	GSN	-
Resolute [Gacek *et al.*(2014)] https://github.com/smaccm/smaccm	Unique notation	Distributed real-time embedded systems
SafeEd [Groza and Marc(2014)] http://cs-gw.utcluj.ro/~adrian/tools/safed/gsn/gsn.html	GSN	-
Safety.Lab [Ratiu *et al.*(2015)]	GSN	-
SAM [Kelly and McDermid(2001a)]	GSN	-
SCT: Safety Case Toolkit [Aiello *et al.*(2014)] http://www.dependablecomputing.com/	GSN, MDD (MultiMarkdown doc.)	-
SBVR/GSN Editor [Luo *et al.*(2015b)]	GSN	-
TurboAC [TurboAC(2018)] http://www.gessnet.com/	Subset of GSN, Tabular, Narrative	Medical devices
Visio add-on [Larrucea *et al.*(2017)] http://www-users.cs.york.ac.uk/~tpk/gsn/	GSN	-

Table 6.2: General tool information.

6.4.3 Evaluation of the Tools and Discussion

Each tool has been manually evaluated for its support in the previously established categories, with the results shown in Table 6.3. Figure 6.15 represents the overall grade distribution for each category. To simplify visualization, all split grades have been rounded up and represented as the higher grade.

Creation. Support for creation of ACs primarily ranges between minimal (43%) and moderate (49%) (see Figure 6.15(a)). The notable exceptions, ENTRUST [Calinescu *et al.*(2017)] and Resolute [Gacek *et al.*(2014)], offer strong support by providing the automatic generation of ACs, based on various underlying system and/or behavioral models. As previously mentioned however, these tools are domain specific. Unless modified, their use is confined to the specific underlying architectural languages, models, etc., that they support. To our knowledge, a tool that can automatically generate complete ACs for a broad range of domains is yet to be developed. Based on these observations, it would seem that the benefits obtained by creating a strong dependency between ACs and system models come at the cost of flexibility and generalized usability.

Maintenance. Again, the absolute majority of tools provide either minimal (51%) or moderate (41%) support for maintenance (see Figure 6.15(b)). Tools with moderate support for maintenance often allow the linking of evidence, models and other artefacts to the corresponding AC elements, making it easy to notify the user of the impacts of the change. In turn, ENTRUST [Calinescu *et al.*(2017)] and ETB [Cruanes *et al.*(2013)] offer strong support by automatically reflecting artefact changes on the AC. ETB [Cruanes *et al.*(2013)] allows the incorporation of 3rd party tools for the purpose of generating evidence and logs timestamps of their invocations in order to determine which analyses are out of date with respect to the current development artefacts, re-running those that are not synchronized. ENTRUST [Calinescu *et al.*(2017)] is tightly coupled with the design-time and runtime models of a system. It has the ability to dynamically verify self-adaptive systems at runtime and update their ACs as necessary.

Assessment. Figure 6.15(c) shows that the results for AC assessment are fairly distributed among all levels of support as compared to the other functional categories, with the majority offering moderate support (38%). The highest percentage of strong support (19%) is seen in this category. Unlike creation and maintenance however, 19% of tools offer no support for assessment. Furthermore, no correlation is seen between support for assessment and any other category, implying that assessment is a fairly standalone tool functionality, the support for which is not largely dependent on the other categories.

Collaboration and Reporting. Most of the tools we surveyed offer no support for collaboration (68%) or reporting (57%). A pronounced trend (see Table 6.3) is that tools with support in these categories are usually industrial, such as ASCE [Netkachova *et al.*(2015)], ISCaDE [Larrucea *et al.*(2017)], NOR-STA [Gorski *et al.*(2012)], OpenCERT [Larrucea(2016)] and SCT: Safety Case Toolkit [Aiello *et al.*(2014)]. Perhaps such capabilities are not receiving adequate interest among researchers, and thus are being developed only after tools reach significant maturity, if at all.

Integration. Support for integration is split between moderate (40%) and none (38%). Not a single tool among the ones we evaluated offered strong support, indicating that some manual integration between other assurance lifecycle activities and the ACs is always required. Table 6.3 shows a strong correlation between high support for integration, and high support for AC creation and maintenance. It would appear that a more integrated environment allows tighter coupling between various artefacts, such as system models and evidence, subsequently enabling automation through dependencies. As previously discussed however, the creation of these dependencies might introduce limitations in other aspects.

6.4.4 Threats to Validity

The main threat to validity in our survey results is the completeness of our list of tools and tool information. Even though our search methodology is thorough, it is possible that it did not capture all existing AC tools. As discussed in Section 6.4.1, our evaluation was based only on information found in the corresponding tool's documentation, publications, website and other publically available resources. It is possible that the description of some functionality received a lower grade because it was not adequately described or the relevant resource was unavailable.

6.4.5 Summary

In this section, we reported on a comprehensive identification and a preliminary evaluation of AC tools, comparing them w.r.t. several categories using the available documentation. In the future, we intend to refine our results using deeper analysis, through a systematic evaluation of the tools themselves.

Our experience shows that there is significant room for improvement of the tools in all of the discussed categories. Furthermore, it appears that several categories are interdependent, i.e., high support in one is strongly correlated with high support in another. For example, we expect that improvements in the integration category will significantly benefit other categories such as creation and maintenance. Yet, to the best of our knowledge, there is currently no tool that supports the seamless linking of the various assurance lifecycle processes.

6.5 Chapter Summary

In this chapter, we have presented required material on the ISO 26262 standard and assurance cases, including assurance case notations and tools as part of a survey we conducted and publised in [Maksimov *et al.*(2018)]. We refer to these concepts in the rest of this thesis. Furthermore, the results of the survey we presented demonstrate the need for tool support for assuance case maintenance, a problem we aim to address in this thesis.

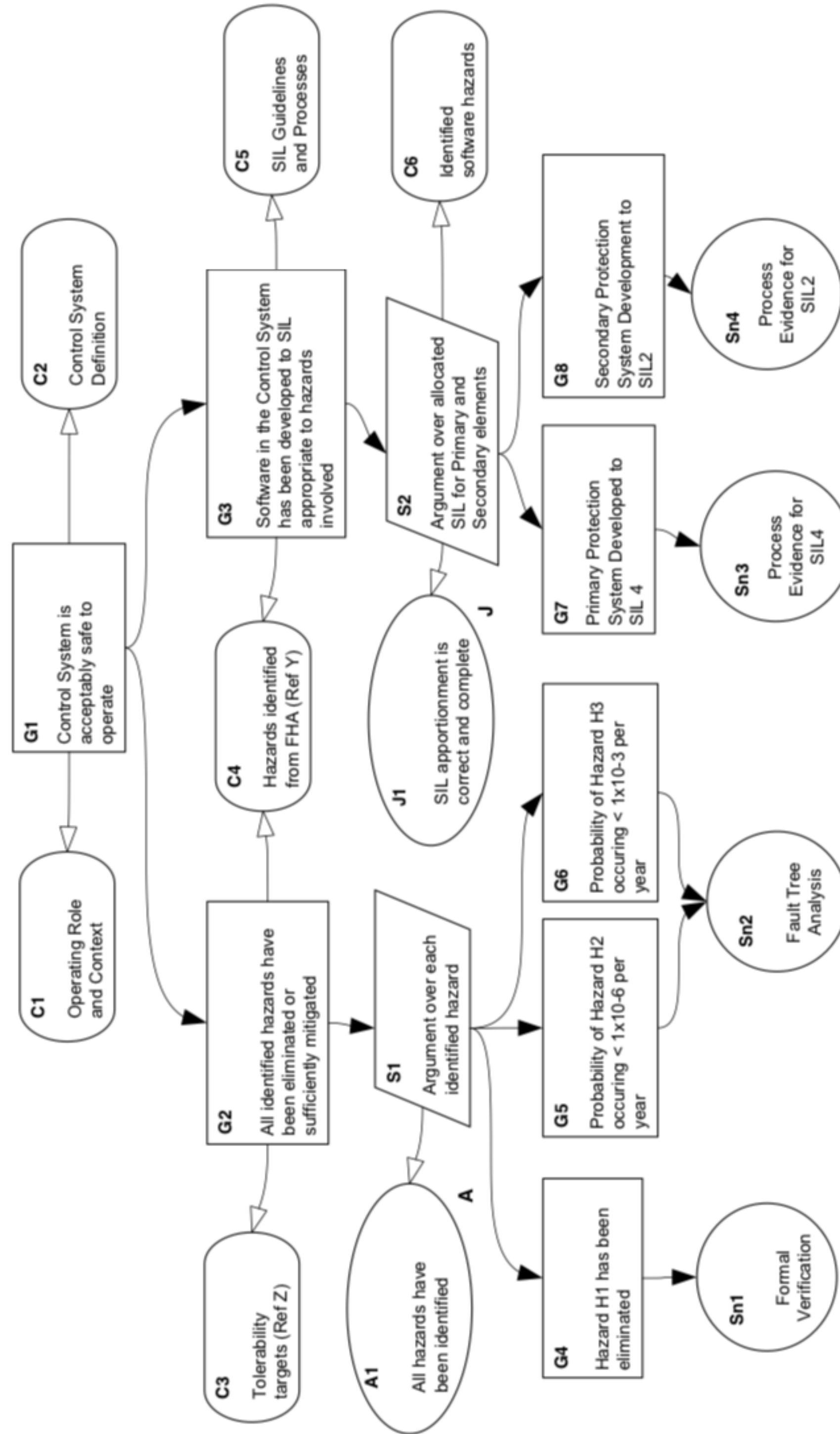


Figure 6.12: Example safety case in GSN from [GSN(2011)].

Claim- a statement asserted within the argument that can be assessed to be true or false, e.g. ▪ "System X is adequately safe during the shut down phase of operation" ▪ "Unit testing is complete" ▪ "All identified hazards are adequately managed in the hazard log" ▪ "Design personnel are suitably qualified" ▪ "Training materials have been reviewed"	Each claim is be supported by a number of sub claims, arguments or evidence. The claim may contain additional contextual material, for example explaining terms used and scope. ASCE contains tools for editing rich narrative at any node in the safety case
Argument- a description of the argument approach presented in support of a claim. e.g.: ▪ "argue by considering safety of subsystems" ▪ "because wiring conforms to relevant electrical standards"	This element is optional, but often it is good practice to include to explain the approach to satisfying the parent claim If the approach to supporting a claim is straightforward or well understood by the intended audience, it is permissible to simply link directly from the supporting claim.
Evidence- a reference to the evidence being presented in support of the claim or argument, e.g. ▪ "the hardware reliability analysis report" ▪ "interlock design documentation"	Usually the evidence node will summarise and link out to the relevant report containing the evidence ASCE contains a number of tools to support: ▪ linking to, management and tracking of changes in the underlying evidence.

Figure 6.13: CAE element definitions and examples.

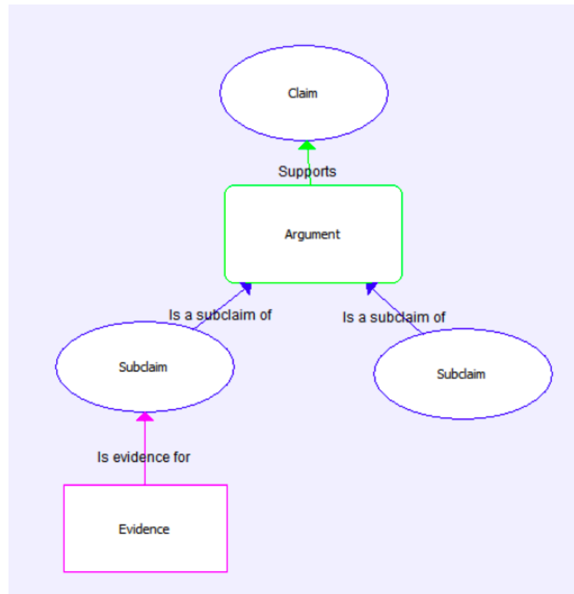


Figure 6.14: An example of a CAE structure.

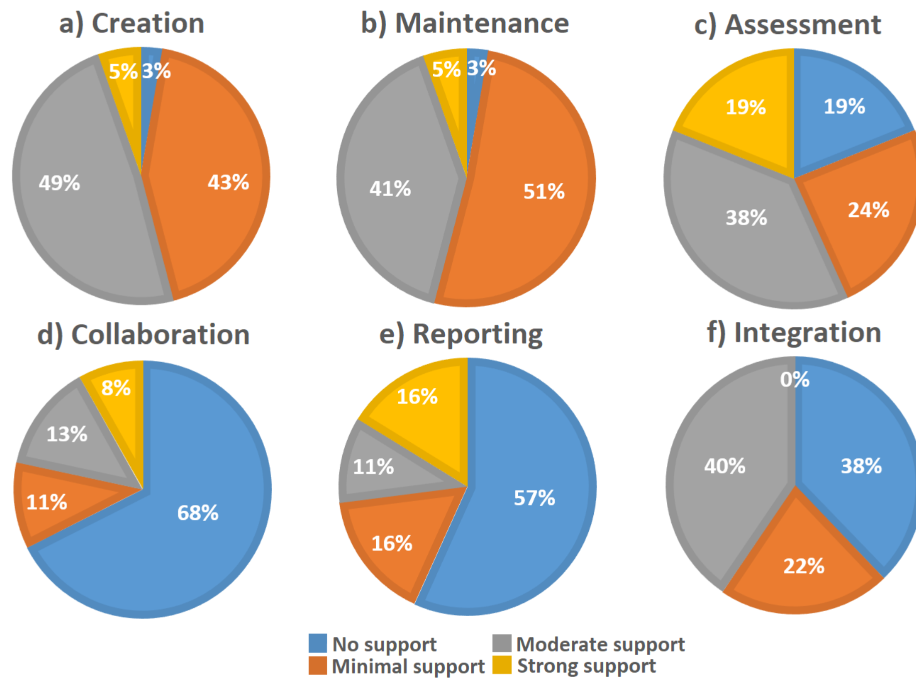


Figure 6.15: Overall AC tool support for: a) creation, b) maintenance, c) assessment, d) collaboration, e) reporting and f) integration.

Tool name	Creation	Maintenance	Assessment	Collaboration	Reporting	Integration
ACBuilder [Kawakami *et al.*(2016)]	B	D	D	D	D	D
ACCESS [Larrucea *et al.*(2017)]	B	C	C	D	C	D
ACEdit [Larrucea *et al.*(2017)]	C	C	B	D	D	D
AdvoCATE [Denney and Pai(2017)]	B	B	A	D	A/B	B
AGSN [Luo *et al.*(2017)]	C	C	B	D	C	D
ASCE [Netkachova *et al.*(2015)]	C	B	B	B	A/B	C
Assure-It [Shida *et al.*(2013)]	C	C/D	D	D	D	D
Astah GSN [Larrucea *et al.*(2017)]	B	C	B	D	C	D
Artisan GSN modeler [University of York(2018)]	B	C	B	A	D	D
AutoFOCUS3 [Cârlan *et al.*(2017)]	B	B	B	D	D	B
CertWare [Barry(2011)]	B	B	A	C	D	B
D-Case Communicator [Matsuno(2017)]	C	C	D	C	D	D
D-Case Editor [Matsuno *et al.*(2010)]	B	B	B	D	D	B
D-Case Weaver [Fujita *et al.*(2012)]	C	C	C	C	C	B
D-MILS [Cimatti *et al.*(2015)]	B	B	B	D	D	B
Eclipse & Papyrus Ext. [Huhn and Zechner(2009)]	C	C	A	D	D	D
eDependabilityCase [Lautieri *et al.*(2004)]	C	C	B	D	D	D
ENTRUST [Calinescu *et al.*(2017)]	A	A	C	D	D	B
eSafetyCase Toolkit [Newton and Vickers(2007)]	B	C	B	B	B	D
ETB (Evidential Tool Bus) [Cruanes *et al.*(2013)]	C	A	D	C	D	B
Event-B extension [Laibinis *et al.*(2015)]	B	C	B	D	D	B
EviCA [Nair *et al.*(2015a)]	C	C	B	D	D	D
GAGE [Bjornander *et al.*(2012)]	D	B	B	D	D	D
HiP-HOPS extension [Retouniotis *et al.*(2017)]	B	B	D	D	D	B
ISCaDE [Larrucea *et al.*(2017)]	B	C	C	B	A/B	B
MMINT-A [Fung *et al.*(2018)]	C	B	C	D	D	C
Modus [Sabetzadeh *et al.*(2013)]	C	B	A	B	C	C
NOR-STA [Gorski *et al.*(2012)]	B	B	B	A	A/B	C
OpenCERT [Larrucea(2016)]	B	B	C	B	B	C
Resolute [Gacek *et al.*(2014)]	A	B	A	D	C	B
SafeEd [Groza and Marc(2014)]	C	C	A	D	B	C
Safety.Lab [Ratiu *et al.*(2015)]	C	B	A/B	D	D	B
SAM [Kelly and McDermid(2001a)]	B	B	C	D	B	B
SCT: Safety Case Toolkit [Aiello *et al.*(2014)]	B	C	C	A/B	A	C
SBVR/GSN Editor [Luo *et al.*(2015b)]	C	C	D	D	D	C
TurboAC [TurboAC(2018)]	B	C	C	D	A/B	B
Visio add-on [Larrucea *et al.*(2017)]	C	C	D	D	D	D

Table 6.3: Evaluation of capabilities of individual tools.

Chapter 7

An Approach for Assurance Case Reuse due to System Evolution

Evolution in software systems is a necessary activity that occurs due to fixing bugs, adding functionality or improving system quality. Systems often need to be shown to comply with regulatory standards. Along with demonstrating compliance, an artifact, called an assurance case, is often produced to show that the system indeed satisfies the property imposed by the standard (e.g., safety, privacy, security, etc.). Since each of the system, the standard, and the assurance case can be presented as a model, this chapter discusses the extension and use of traditional model management operators to aid in the reuse of parts of the assurance case when the system undergoes an evolution. Specifically, we present a model management approach that eventually produces a partial evolved assurance case and guidelines to help the assurance engineer in completing it. We demonstrate how our approach works on an automotive subsystem regulated by the ISO 26262 standard. The content of this chapter has been published in [Kokaly *et al.*(2016a)].

7.1 Introduction

The pervasiveness of software in all aspects of human activity has created special concerns regarding issues such as safety, security and privacy. Governments and standard organizations (e.g., ISO) have responded to this trend by creating regulations and standards that software must comply with. For companies, compliance is a complex and costly goal to achieve. They may have to comply with multiple standards due to multiple jurisdictions and track the changes in standards. *Assurance cases* – collections of arguments and evidence to support the claims of compliance – must be developed and managed. Finally, maintaining families of related software products further multiplies the effort. Increasingly, models and model-driven engineering are being used as means to facilitate communication and collaboration between the stakeholders in the compliance value chain

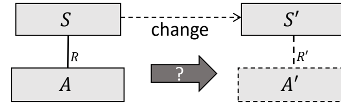


Figure 7.1: Assurance case evolution scenario.

and further to introduce automation into regulatory compliance tasks.

In Chapter 1, we a laid out a research agenda for applying model management to address the software compliance problem and sketched its use in particular compliance management scenarios. In this chapter, we focus on one of these scenarios – assurance case reuse due to system evolution (**P2**) – and develop it in detail. Figure 7.1 illustrates the scenario at a high level. Assume that a current specification S describes the specification for the software in a vehicle. In addition, a type of assurance case A, called a *safety* case, has been developed complying with the ISO 26262 vehicle functional safety standard [ISO(2011)]. Safety case A contains perhaps thousands of safety claims about different components of the vehicle, as well as arguments and evidence to support these claims. Now if S is evolved to S' – for example, as a result of a new requirement or a bug fix – a corresponding safety case A' for S' must be developed. Due to complexity and effort required to develop a safety case, there is strong incentive to reuse as much of A as possible in the creation of A'. We address this problem using a model management strategy.

Contributions. This chapter makes the following contributions:

1. We define a generic model management framework for assurance case reuse due to model evolution.

2. We identify and specify the model management operators needed for a semi-automated solution to the assurance case reuse problem and present an algorithm for reuse.

3. We evaluate the generic framework and proposed solution by instantiating it for ISO 26262 vehicle safety cases with the KAOS goal modeling language [Dardenne *et al.*(1993)] used for expressing assurance cases. We then apply this instantiation to an automotive subsystem, namely, a power sliding door system.

Organization. The rest of this chapter is structured as follows: Section 7.2 discusses our generic assurance case reuse framework, where we present an algorithm that is then evaluated in Section 7.3. Section 7.4 is an application of our generic framework on the power sliding door example. Section 7.5 discusses related work, and Section 7.6 ends with a chapter summary.

7.2 A generic assurance framework for model evolution

In this section, we develop a generic model management-based framework for assurance case reuse in the context of system evolution.

7.2.1 Objective of Reuse

The objective of the assurance case reuse problem due to the system evolution can be stated as follows:
(Objective) Given system specification S with complete and correct assurance case A, if S evolves to S', determine the *maximal reuse* of A to produce a complete and correct assurance case A' for S'.

Note that we take a "complete and correct assurance case" to mean one that is *acceptably* complete and correct by the organization developing the system. The goal of finding the maximal reuse can be refined into two subgoals:

1. identify the *impact set* $A_{S-S'}$ – the subset of A impacted by the change in S; and
2. identify the *kind* of impact for the atoms within the impact set.

Goal 1 implies that atoms of A outside the impact set can be reused within A' since they are not impacted by the change; thus, the impact set implicitly defines the maximal subset of A that can be reused. The relevance of Goal 2 is that there are two possible types of impact to an atom due to a change, and this affects the degree of reuse:

1. The change may affect the truth state of a claim or the validity of a piece of evidence. Thus, the claim/evidence can be reused directly but its truth/validity state must be rechecked.
2. The change may affect the definition of a claim, argument or piece of evidence and hence affect its interpretability. Thus, the claim, argument or piece of evidence must first be revised and then additionally, in the case of claim/evidence, its truth/validity state must be checked.

A Type 1 impact requires less effort because the claim/evidence can be reused directly (i.e., no revision) and rechecking can sometimes be automated. For example, rechecking a test-based evidence involves re-running the test cases, rechecking a claim containing a formally specified property may be rechecked using a property checker, etc. Thus, a Type 1 impact exhibits greater reuse than a Type 2 impact.

For example, assume that the assurance case A contains the claim that the following property holds for the AutoLight subsystem that controls a vehicle headlight:

(Property1) If the ambient light sensor detects less than 25 lumens then the head lights turn on.

Furthermore, assume that this claim has been verified using a set of test results as evidence. Now if the subsystem is evolved so that it uses a new algorithm to turn the head lights on, then this

claim is impacted because its truth state may be affected, and the property needs to be rechecked. Furthermore, the evidence used previously is no longer valid and the tests must be re-performed to check the property. Now consider another evolution of the subsystem in which the ambient light sensor is removed since some other approach to detecting light is used. In this case, the definition of the claim is affected and the property Property1 can no longer be properly interpreted since there is no ambient light sensor. In this case the claim must be first revised based on the new design to be interpretable and then have its truth state checked. Similarly, a revision to the test cases producing the evidence is required.

Identifying the impact set and kinds of impacts represents the *ideal* behaviour for an assurance case impact assessment approach. We define an actual impact assessment approach as follows:

Definition 23 (Impact assessment approach) *Given a system specification S with a complete and correct assurance case A, if S evolves to S' then an* impact assessment approach *R can be applied as $R(S, A, S')$ to produce a pair $\langle A_R, k\rangle$ where A_R is the* impact set estimate *and function $k : A_R \rightarrow \{\texttt{revise}, \texttt{recheck}\}$ identifies the* impact kind annotation *for the atoms of A_R.*

We can evaluate an impact assessment approach by comparing it to the ideal case.

Definition 24 (Soundness and relative efficiency) *Given a system specification S with a complete and correct assurance case A, and S evolves to specification S',*
(Soundness) Impact assessment approach R is sound *if*
$A_{S-S'} \subseteq A_R$.
(Relative efficiency) A sound impact assessment approach R is relatively more efficient than *R' iff*
$A_R \subseteq A'_R$.

Soundness is a *correctness* criterion for an impact assessment approach – a sound approach guarantees that all actually impacted atoms of the assurance case are identified by the approach. Relative efficiency is a *quality* criterion, and greater efficiency means that it finds fewer "false positives" (i.e., when it says an atom is impacted although it is not) and hence facilitates greater reuse.

7.2.2 The Framework

We have defined the objective of reuse and how to evaluate a possible impact assessment approach. We now define a generic assurance case evolution framework RMM (Reuse with Model Management) based on model management that addresses reuse. The framework is *generic* because, as is typical with model management, it is defined independently of the specific model types used in an evolution scenario. Thus, applying it to a particular evolution case requires instantiating it for the model types used.

Figure 7.3 gives a conceptual overview of the framework as well as embedding the impact assessment algorithm detailed below. The initial systems specification S has a corresponding assurance case A. These are connected by traceability relation R. System S is first evolved by changing it to a system specification S'. The difference between these specifications is captured in the relation

D. After performing the impact assessment algorithm, the resulting impact set estimate $A_{\texttt{RMM}}$ and impact kind annotation $k_{\texttt{RMM}}$ (see Definition 23) are used as guidance by an Assurance Engineer to complete the new assurance case A' and the corresponding traceability relation R'.

Impact assessment algorithm. The impact assessment algorithm used by RMM assumes that potential impact on an assurance case is defined as follows.

Definition 25 (Potential impact) *An atom in A is* potentially impacted *by the change D iff it is dependent on an atom of A that mentions the name/identifier of an atom of S that is itself affected by the change D.*

Potential impact means that the atom *may* be impacted but is not guaranteed to be impacted. For example, it is possible for a claim to be identified as potentially impacted but its truth value happens not to change. However, an atom that does not satisfy this definition *is* guaranteed to be unimpacted.

The algorithm makes the following assumptions:

Assumption 3 (RMM Assumptions)

3.1. Specifications S and S' consist of one or more related models defining the systems. We identify the type of these specifications as T.

3.2. Delta D consists of the three submodels $C0a \subseteq S'$, $C0d \subseteq S$ and $C0c \subseteq S'$ representing the added atoms, deleted atoms and changed atoms, respectively. Atom addition and deletion can apply to any element, reference or attribute. Atom changes are limited to attribute value changes and changes in the target element of a reference.

3.3 Assurance case A is considered acceptably correct and complete by the organization developing the system.

3.4 We are provided with a correct model slicer $\texttt{Slice}_T$ and merge operator $\texttt{Merge}_T$ for models of type T. In particular, $\texttt{Slice}_T$ is assumed to identify all atoms affected by the slicing criterion.

3.5. Traceability relation R links an atom $x \in A$ to an atom $y \in S$ iff x mentions the name or identifier of y, i.e., x makes a direct reference to y.

Thus, in Definition 25, checking whether an atom of A mentions an atom of S is possible via R due to Assumption 3.5 and checking whether an atom of S is affected by D is possible using $\texttt{Slice}_T$ due to Assumption 3.4. In addition, checking whether an atom of A is dependent on another atom of A is done using the relation $ACdep$ (see Definition 22).

To determine kinds of impacts on atoms we observe that if an impacted atom of A mentions an atom of S that is deleted, it must be revised because it can no longer be "well-formed". In addition, any other atoms of A that are $ACdep$ dependent on the revised atom *may* need revision as well. All other impacted atoms need not be revised but need to be rechecked. Note that this focuses on deletion as the sole cause of revision. Clearly, additions of atoms in S' likely lead to revisions on A as well, but since these are new, the places where such revisions occur cannot be detected by the

impact assessment algorithm; thus, these decisions are left for the post-algorithm manual step by the Assurance Engineer.

The algorithm is given in Figure 7.2. This a *model management* algorithm, i.e., it is expressed in terms of standard types of model management operators. In addition, the algorithm is expressed at an abstract level and is parameterized by operators for model type T of the specification S. These operators are given a parameters $\texttt{Slice}_T$ and $\texttt{Merge}_T$. Figure 7.3 gives a visual overview of the algorithm embedded in the overall RMM evolution process. The gray circled numbers correspond to line numbers in the algorithm.

In line 1 of the algorithm, the traceability map from S' to A is computed by restricting the original map R using delta D (see Section 7.2.3). Thus R'_A contains the mappings from all the atoms in S' except the added ones. Lines 2 and 3 use the T-specific slicing operator ($\texttt{Slice}_T$) to expand the changed regions to all affected atoms of S and S', respectively. In line 4, these are then traced across the traceability relations to A and merged (see Section 7.2.3) to identify the core subset $C2_{\text{recheck}}$ of A that must be rechecked. In line 5, the core subset $C2_{\text{revise}}$ of A that must be revised is obtained by tracing the set of deleted atoms $C0d$ across the traceability relation. In lines 6 and 7, these core subsets are expanded to the full impacted subsets using the assurance case slicing operator (see Section 7.2.3) Finally, in lines 8-11, the impact set estimate $A_{\texttt{RMM}}$ and impact kind annotation $k_{\texttt{RMM}}$ are prepared and returned as the output of the algorithm.

Post-algorithm human actions. After applying the impact assessment algorithm, the Assurance Engineer uses the impact set and impact kind annotation on A as guidance to completing the new assurance case A' for S'. All atoms in A not in $A_{\texttt{RMM}}$ are considered unimpacted and can be reused in A' without change. Those atoms marked 'revise' must be changed to make them well-formed and then rechecked. The action for an atom marked 'recheck' is dependent on its type. A Claim must be re-evaluated to check that it has an acceptable truth state (e.g., *true* or *affirmed*) based on the arguments that support it. For an Evidence element, the procedure for producing the results must be re-performed. For example, if the evidence consists of test results, the test cases must be rerun; for analysis results, the analysis procedure is rerun, etc. In some case the recheck procedure may be automated. For an Argument, its details must be checked to ensure that they are still valid. If the result of the recheck is not acceptable (e.g., new test results fall outside acceptable limits) the Assurance Engineer must assess how to respond. For example, this may mean that the atom must now be revised or it may mean that the system specification requires further changes.

7.2.3 Additional Model Management Operators

Other than the operators $\texttt{Slice}_T$ and $\texttt{Merge}_T$ provided as parameters, the algorithm in Figure 7.2 uses four additional model management operators. We describe them declaratively below.

Definition 26 (Additional Operators) *Let R be the traceability relation between S and A, $D = \langle C0a, C0d, C0c\rangle$ be the delta between S and S'. Then*

Algorithm: `RMM` **impact assessment**
Params: $\langle \texttt{Slice}_T, \texttt{Merge}_T \rangle$
Input: initial spec $S : T$, assurance case $A : AC$,
traceability map R, changed spec $S' : T$,
delta $D = \langle C0a, C0d, C0c \rangle$
Output: Impact set estimate $A_{\texttt{RMM}}$, impact kind annotation $k_{\texttt{RMM}}$

1: $R'_A \leftarrow \texttt{Restrict}(R, D)$
2: $C1dc \leftarrow \texttt{Slice}_T(S, \texttt{Merge}_T(C0d, C0c))$
3: $C1ac \leftarrow \texttt{Slice}_T(S', \texttt{Merge}_T(C0a, C0c))$
4: $C2_{\text{recheck}} \leftarrow \texttt{Merge}_{AC}(\texttt{Trace}(R, C1dc), \texttt{Trace}(R'_A, C1ac))$
5: $C2_{\text{revise}} \leftarrow \texttt{Trace}(R, C0d)$
6: $C3_{\text{revise}} \leftarrow \texttt{Slice}_{AC}(M, C2_{\text{revise}})$
7: $C3_{\text{recheck}} \leftarrow \texttt{Slice}_{AC}(M, C2_{\text{recheck}})$
8: $A_{\texttt{RMM}} \leftarrow \texttt{Merge}_{AC}(C3_{\text{revise}}, C3_{\text{recheck}})$
9: $k_{\texttt{RMM}}(C3_{\text{recheck}}) \leftarrow$ '`recheck`'
10: $k_{\texttt{RMM}}(C3_{\text{revise}}) \leftarrow$ '`revise`'
11: **return** $A_{\texttt{RMM}}, k_{\texttt{RMM}}$

Figure 7.2: Algorithm for assessing assurance case impact due to system evolution used in the `RMM` evolution framework.

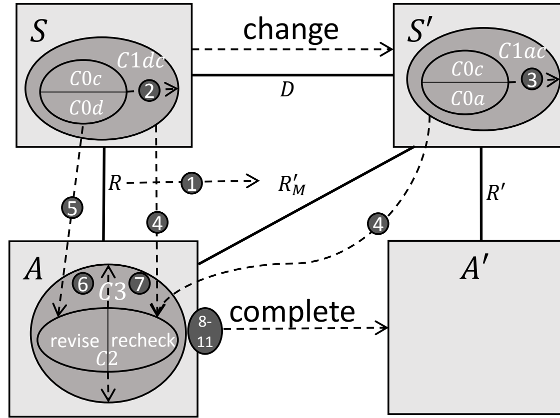


Figure 7.3: Conceptual overview of model management based assurance case evolution framework RMM. Numbers in gray circles correspond to the line numbers of the impact assessment algorithm in Figure 7.2.

- $\texttt{Restrict}(R, D)$ *is the relation between* S' *and* A *defined as* $\{\langle a, c \rangle \mid a \in atoms_A \wedge c \in (atoms_{S'} \setminus atoms_{C0a}) \wedge R(a, c)\}$.
- $\texttt{Trace}(R, C)$*, where* $C \subseteq atoms_S$*, is the subset of* $atoms_A$ *defined as* $\{a \mid a \in atoms_A \wedge \exists c \in atoms_C \cdot R(a, c)\}$.
- $\texttt{Slice}_{AC}(A, C)$*, where* $C \subseteq atoms_A$*, is the subset of* $atoms_A$ *defined as* $\{a' \mid a \in C \wedge$

$ACdep(a, a')\}$.

- $\texttt{Merge}_{AC}(A, A')$ *is the assurance case* $A'' : AC$ *defined as* $atoms_{A''} = atoms_A \cup atoms_{A'}$.

$\texttt{Restrict}(R, D)$ creates a new traceability relation from S' to A that contains all links in R except those involving deleted atoms. $\texttt{Trace}(R, C)$ traverses the traceability relation R to produce the set of atoms linked to atoms in C. $\texttt{Slice}_{AC}(A, C)$ expands a subset C of atoms in A to the subset of all atoms dependent on atoms in C through the $ACdep$ relation defined in Definition 22. $\texttt{Merge}_{AC}(A, A')$ produces the assurance case containing the union of atoms in each of A and A'. Note that in our algorithm we use $\texttt{Merge}_{AC}$ to combine submodels of a larger model. If A and A' overlap, the union includes only one copy of the atoms in the overlap.

7.3 Algorithm Analysis

Definition 24 provided two criteria against which an assurance case impact assessment approach can be evaluated. In this section, we apply these criteria to the impact assessment algorithm in Figure 7.2 and briefly discuss the important issue of *emergent propeties* [Johnson(2006)] in systems and how the algorithm handles this.

7.3.1 Soundness

As discussed in Section 7.2.2, although the algorithm does find some indirect impacts due to added atoms in $C0a$, it cannot possibly find them all since the Assurance Engineer is required to assess these. Thus, we limit our soundness claim to evolution due to atom changes and deletions.

Proposition 1 *Given a system specification* S *with an assurance case* A *and a traceability relation* R*, and* S *evolves to a specification* S' *with delta* $D = \langle C0a, C0d, C0c \rangle$*, the impact assessment algorithm in Figure 7.2 is sound as defined in Definition 24 with respect to impacts due to atoms in* $C0d$*,* $C0c$*.*

(Proof) We use Definition 25 as the definition of potential impact and argue by contradiction. Assume the algorithm is not sound. Then there is an atom $x \in atoms_A, x \notin A_{\texttt{RMM}}$ and yet it is impacted by some atom $y \in D$. Since x is impacted by y, according to Def 25, this means that either (1) x directly mentions y, (2) x mentions some other atom y' in S' or S that is affected by y, or (3) x is dependent on another atom x' in A that is impacted by y.

In case (1), x will necessarily be in one of $C0c$ or $C0d$ (added atoms cannot be mentioned in A) and so it will be in $C1dc$ in line 2. Also, if a mentions x then it will be in $C2_{\text{recheck}}$ or $C2_{\text{revise}}$ due to Assumption 3.5 and the use of $\texttt{Trace}$ in lines 4-5. The slicing in lines 6-7 will retain x since $ACdep$ is reflexive and thus $x \in A_{\texttt{RMM}}$ is in contradiction to our assumption. Case (2) is similar to case (1) except that we use Assumption 3.4 to ensure that if y' is affected by y then it is captured by $\texttt{Slice}_T$ in lines 2-3. Also, we allow x to be an added atom in $C0a$ since added atoms can indirectly

impact atoms of A in lines 3-4. Finally in case (3), if x' is impacted, then x' must be in $C3_{\text{recheck}}$ or $C3_{\text{revise}}$ constructed using $\texttt{Slice}_A C$ in lines 6-7. But since $ACdep$ is transitive and $ACdep(x, x')$ holds, $\texttt{Slice}_{AC}$ would capture x in line 6 or 7 and then $x \in A_{\texttt{RMM}}$ in line 8 is in contradiction with our assumption. Since we have shown that all cases contradict the assumption that $x \notin A_{\texttt{RMM}}$, we conclude that if x is impacted by y then $x \in A_{\texttt{RMM}}$, and so the algorithm is sound.

7.3.2 Relative Efficiency

According to Definition 24, an impact assessment approach is more efficient if it reports fewer "false positives". Because RMM is defined at an abstract level, its efficiency is determined by the information it has available on which to base impact assessments. For example, the algorithm will mark a claim x to be rechecked if it is $ACdep$ dependent on another claim x'. However, if we had access to the particular argument structure connecting the claims, it may be that changing the truth state of x' does not affect the truth state of x and so x should not have been marked for recheck. In Chapter 8 we discuss ways that the efficiency could be improved by utilizing additional information available when the framework is instantiated for a particular modeling language.

7.3.3 Emergent Properties

An emergent property of S arises as a result of the integration of parts of S, where no part of the system is directly responsible for it. We consider two cases of emergent properties: *system properties* [Leveson(1995)] and *feature interactions* [Calder *et al.*(2003)].

Consider the following claim for a vehicle: "99.5% of the time, a collision when the vehicle is moving 80kph will not result in a fatality of a passenger." In an assurance case for the vehicle system, the argument to support this claim might use crash test results as evidence. This property, if true, clearly results from the interaction of the parts of the vehicle rather than from any one part. Evolving the vehicle specification may impact the truth of this claim but some changes would not. For example, a change to the headlight colour would probably have no impact on the truth of the claim. In this scenario, the RMM impact assessment algorithm behaves conservatively – if the vehicle specification evolved *in any way*, the claim is flagged to be rechecked. This follows because we would expect the specification slicer $\texttt{Slice}_T$ to identify the whole system "vehicle" to be affected by any change within the specification, and the identifier "vehicle" is mentioned directly in the claim. Thus, although the RMM assessment algorithm sacrifices efficiency by being conservative, it ensures soundness for such system properties.

A feature interaction occurs when using two or more features together results in an unintended behaviour. For example, assume the AutoLight subsystem described above interferes with the AutoOff subsystem responsible for (among other things) turning off the headlights when they are left on after the car is turned off. Even if an assurance case contains separate claims about each subsystem, since they interact, a change to one subsystem could impact the claim of the other. Since the RMM algorithm relies on the slicer $\texttt{Slice}_T$ to detect dependencies between parts of the specification, the extent to

which impacts due to feature interaction are handled correctly depends on the quality of the slicer (see Assumption 3.4). Thus, the algorithm is sound in these situations "up to" the soundness of the slicer.

7.4 Demonstration: PSD example

In this section, we demonstrate our general approach for the reuse of assurance case artifacts on an automotive subsystem, namely, the PSD system introduced in Section 2.1, which is shown to be compliant with part of the ISO 26262 standard. First, we instantiate the framework for specific models of the system and the assurance case. Afterwards, we present the example along with the application of our framework on an evolution scenario.

7.4.1 Instantiating the Framework

For the purpose of the example presented here, we instantiate our general framework such that its input is an initial specification (S) of a system given by a megamodel [Diskin *et al.*(2013)] comprised of a class diagram, a sequence diagram and a relationship between them. This megamodel forms the type (T) of our system specification. The assurance case A for the initial system is given by a KAOS goal tree model (AC), along with traceability to the system megamodel. We are also given an evolution scenario that creates an evolved specification (S') along with a mapping from the original specification (D). We assume we are given class diagram slice and merge operators, similar to those presented in [Lano and Rahimi(2010)] and [Fahrenberg *et al.*(2014)], respectively. We also assume we are given sequence diagram slice and merge operators similar to those presented in [Noda *et al.*(2009)] and [Widl *et al.*(2012)], respectively. Finally, we assume that the slice and merge operators, $\texttt{Slice}_T$ and $\texttt{Merge}_T$, respectively, for the megamodel comprised of the class diagram and sequence diagram and the relationship between them can be computed and are given. The megamodel slice is then computed by following the approach presented in Chapter 5.

7.4.2 Application to PSD System

Consider our running example. Following the guidelines of ISO 26262, we consider the following hazard which is obtained via appropriate hazard analysis techniques [ISO(2011)]: **HE1**: "the activation of the actuator while driving at a speed above 15 km/h, with or without a driver request.". Then, a safety goal is presented in such a way as to prevent the hazard from occurring: **SG1**: "Avoid activating the actuator while the vehicle speed is greater than 15 km/h.".

Once we have a safety goal defined, a set of functional safety requirements **FSR1-5** are put in place to help achieve **SG1**[1]. **FSR1** states that the *VS ECU* sends accurate vehicle speed information to the *AC ECU*. Alternatively, this means that the incorrect transmission that the vehicle speed is

[1] For simplicity, we ignore the ASIL (Automotive Safety Integrity Level) assignments and decomposition and address these later in Chapters 8 and 9.

less than or equal to 15 km/h is prevented. **FSR2** states that the *AC ECU* does not power the *Actuator* if the vehicle speed is greater than 15 km/h. **FSR3** is a requirement that the *VS ECU* sends accurate vehicle speed information to the *Redundant Switch*. **FSR4** ensures the *Redundant Switch* is in an open state if the vehicle speed is greater than 15 km/h. And finally, **FSR5** ensures that the *Actuator* operates only when powered by the *AC ECU* and the *Redundant Switch* is closed.

Figure 7.4 shows a KAOS goal tree that depicts the refinement of **SG1** into **FSR1-FSR5**. This refinement is based on the AND-refinement strategy. For simplicity, we skip the TSRs which are decomposed into HWSRs and SWSRs (see Section 6.2.3), and we go directly from the FSRs to the evidence nodes shown in ellipses. For **FSR1**, since this is a requirement related to the quality of the vehicle speed sensor, evidence is given by some test results performed on the sensor. For **FSR2-5**, these can be specified as properties that we can check on the system model using some model-checking tool. Moreover, **SG1** and **FSR1-FSR5** are all expressed in temporal logic, and it can be proven that (**FSR1** $\wedge$ **FSR2** $\wedge$ **FSR3** $\wedge$ **FSR4** $\wedge$ **FSR5**) $\vdash$ **SG1**. Note that traceability to the system model is given by referencing the parts of the goals that appear in the system model in black font.

7.4.3 Evolution of PSD System

Up to this point, we have a description of our system specification S which is of type "megamodel of class diagram and sequence diagram" T, safety case A which is of type $KAOS$ (recall that a safety case is a particular kind of assurance case), relationship R between S and A, and we are given slice and merge operators for T ($\texttt{Slice}_T$ and $\texttt{Merge}_T$) as discussed in Section 7.4.1. We now describe an evolution scenario and demonstrate how our algorithm works on it.

Consider that the power sliding door system changes in order to decrease its integrity (change in model quality). This could be due to the need to minimize costs and produce a cheaper vehicle. The redundancy is therefore eliminated and the *Redundant Switch* is removed, as shown in Figure 7.5, also borrowed from Part 10 of ISO 26262.

The dynamic *VS ECU* provides the *AC ECU* with the vehicle speed. The *AC ECU* monitors the driver's requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so commands the *Actuator*. The *Actuator* is activated when it is powered.

The parts with labels underlined in each of Figure 2.2 and Figure 2.3 represent the delta D. All the underlined parts of the class diagram and the underlined parts in the top thread of the sequence diagram are to be deleted ($C0d$); however, the underlined parts in the guards of the bottom two threads of the sequence diagram mean that these guards will be changed ($C0c$). In this example, no new parts are added in S', so $C0a$ is empty. S' is the parts of S without the underlined components in each of the class and sequence diagram.

Having all of the required parameters ($\texttt{Slice}_T, \texttt{Merge}_T$) and inputs to the algorithm (initial spec $S : T$, assurance case $A : KAOS$, traceability map R, changed spec $S' : T$ and delta $D = \langle C0a, C0d, C0c \rangle$), we now demonstrate application of the reuse algorithm presented in Figure 7.2.

Line 1 produces the traceability of the assurance case to the evolved system based on the changes made. This is shown in Figure 7.6 by colouring the elements that reference the changed components in black.

Lines 2-3 use the specific slicers and merge operators for our megamodel as described in Section 7.4.1 to expand the changed regions to all affected elements of S and S'. This means deleting or changing elements based on deleted ($C0d$) and changed ($C0c$) elements, respectively. For example, the removal of the *Redundant Switch* will impact the behaviour of the *Actuator* and therefore, its *powered* and *activated* states. But the relation between the *Actuator* and *VS ECU* is not impacted by the removal of the *Redundant Switch*, which means the speed reading given to the *Actuator* is not impacted. This is something we would get from the system-level change impact analysis, and we assume that $\texttt{Slice}_T$ is powerful enough to catch that.

Line 4 identifies the core subset of the original assurance case A that must be rechecked by tracing from the results of Lines 2-3 back to the original assurance case A. $C2_{\text{recheck}} = (\mathbf{FSR2}, \mathbf{E2})$.

Line 5 identifies the core subset of original assurance case A that must be revised by tracing the set ($C0d$) across the traceability relation between S and A. $C2_{\text{revise}} = (\mathbf{FSR3}, \mathbf{E3}, \mathbf{FSR4}, \mathbf{E4}, \mathbf{FSR5}, \mathbf{E5})$.

Lines 6-7 expand the core subsets identified in lines 4 and 5 to produce the full rechecked/revised subsets. In this example, **SG1** is directly affected by the change and is marked 'revised' on line 5. Yet it could be the case that is does not refer to elements being deleted/changed and is marked 'revised' in line 6 because it is the parent claim of claims that have been marked 'revised', and this would be caught by the assurance case slicer. The results so far implicity mean that the set of resuable components is $= (\mathbf{FSR1}, \mathbf{E1})$.

Lines 8-11 produce the impact set estimate $A_{\texttt{RMM}}$ which is an assurance case given by the KAOS tree in Figure 7.6, along with the kind annotation $k_{\texttt{RMM}}$ which is represented by the following: checkmark means that claim can be reused safely; circular arrow – that claim should be rechecked; exclamation mark – that claim should be revised.

Finally, the Assurance Engineer will take the result of the algorithm, which is the annotated goal tree in Figure 7.6, and make some decisions to produce an evolved safety case. For example, she may decide that the top level safety goal remains the same, and yet she defines a new set of FSRs that will help achieve this safety goal. **FSR1** states that the *VS ECU* sends the accurate vehicle speed information to the *AC ECU* (safely reused **FSR1** from original system). **FSR2** states that the *AC ECU* does not power the *Actuator* if the vehicle speed is greater than 15 km/h (rechecked **FSR2** from the original system and reused it as it still holds). **FSR3** states that the *Actuator* is activated only when powered by the *AC ECU* (revised **FSR5** and removed part about *Redundant Switch*). Note that both **FSR3** and **FSR4** from the original goal tree have been deleted as they were revised and removed since they no longer impact the system or the top level safety goal. A final goal tree for the evolved assurance case is given in Figure 7.7. We can then prove that $(\mathbf{FSR1} \wedge \mathbf{FSR2} \wedge \mathbf{FSR3}) \vdash \mathbf{SG1}$. This is compliant with the ISO 26262 refinement guidelines

presented in Section 6.2.3, as desired.

7.5 Related Work

We identify three main categories of related work: work on model evolution, work on modeling of assurances cases, and work on assurance case reuse due to system evolution. We describe them below.

Model evolution. A survey on the evolution of UML models in model-driven software development is presented in [Khalil and Dingel(2013)]. The scenarios that cause a model to change are discussed and they form the basis for system evolution in our approach. Our approach is consistent and complimentary to the existing work on model evolution, including theoretical work on model synchronization [Diskin *et al.*(2014)]. However, we specifically focus on assurance cases as the target of coevolution due to our interest in compliance management. An assurance case in this sense is not a traditional model describing a system specification, but a model of an argument over the system satisfying a property of interest, which differentiates our work from the traditional work on model evolution.

Modeling of assurance cases A variety of methods have been proposed for modeling assurance cases. Goal models and requirements models are used in [Ghanavati *et al.*(2011)]. [Brunel and Cazin(2012)] presents a formal approach for safety argumentation using KAOS goal models and applies it to a Complex UAV System. The GSN notation [Kelly and Weaver(2004)] has also been proposed as a modeling notation for assurance cases. Our work builds on all of these ideas and assumes that an assurance case can be modelled in a variety of ways as long as it presents the core components – claims, arguments and evidence.

Assurance case reuse due to system evolution. [Fenn *et al.*(2007)] and [Kelly and McDermid(1997)] also approach the problem of safety assessment after design changes. [Fenn *et al.*(2007)] only defines the problem of incremental certification and offers some thoughts on how to address it from the perspective of assurance cases represented in modular GSN diagrams. [Kelly and McDermid(1997)] proposes the notion of patterns for building GSN diagrams. Our model-based approach to assurance case reuse could use the structure of such patterns to identify which parts of the GSN diagram can be reused.

7.6 Chapter Summary

In this chapter, we have presented a generic framework for the reuse of assurance case components due to system evolution. We specified a model management reuse algorithm which uses known model management operators (e.g., slice, merge) and produces a semi-automated solution to the assurance case reuse problem. We evaluated our algorithm and demonstrated its applicability on an automotive subsystem – a power sliding door system.

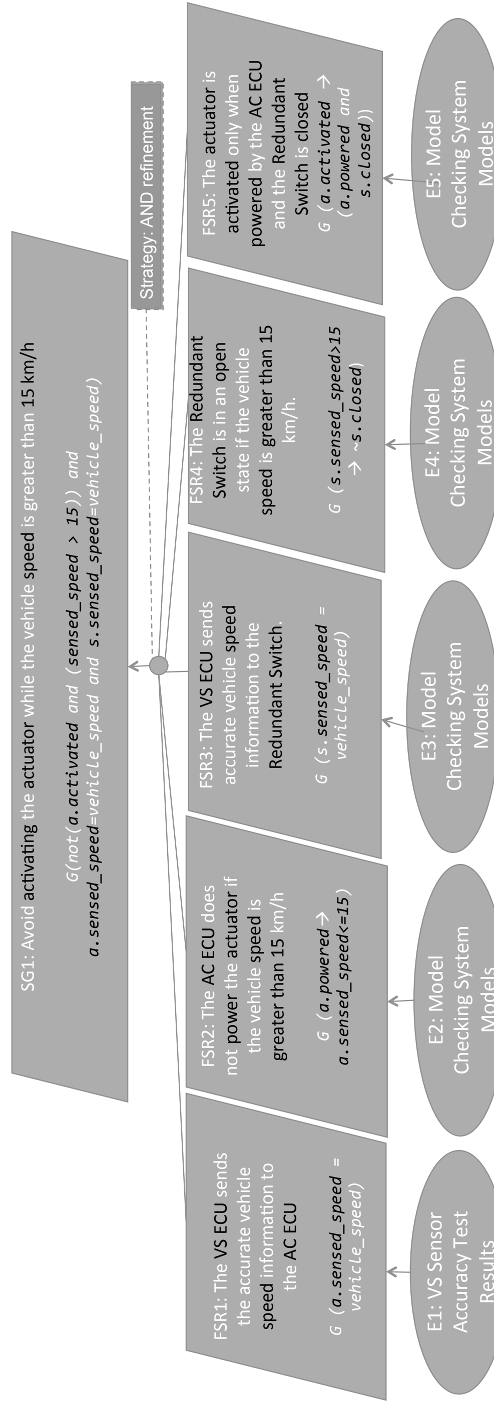


Figure 7.4: Goal tree for system with redundancy.

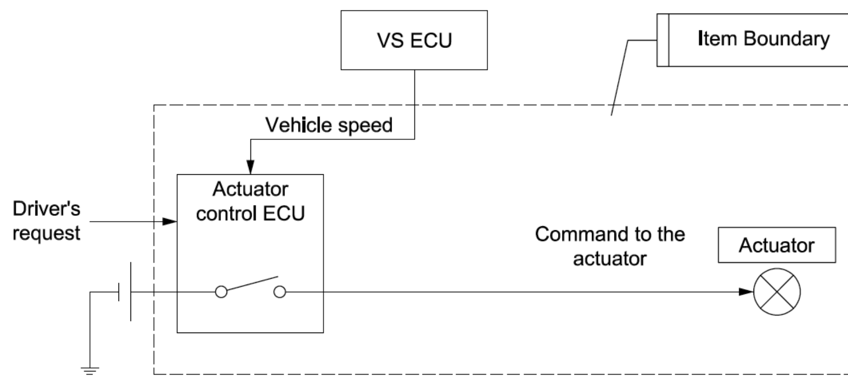


Figure 7.5: PSD system without redundancy [ISO(2011)].

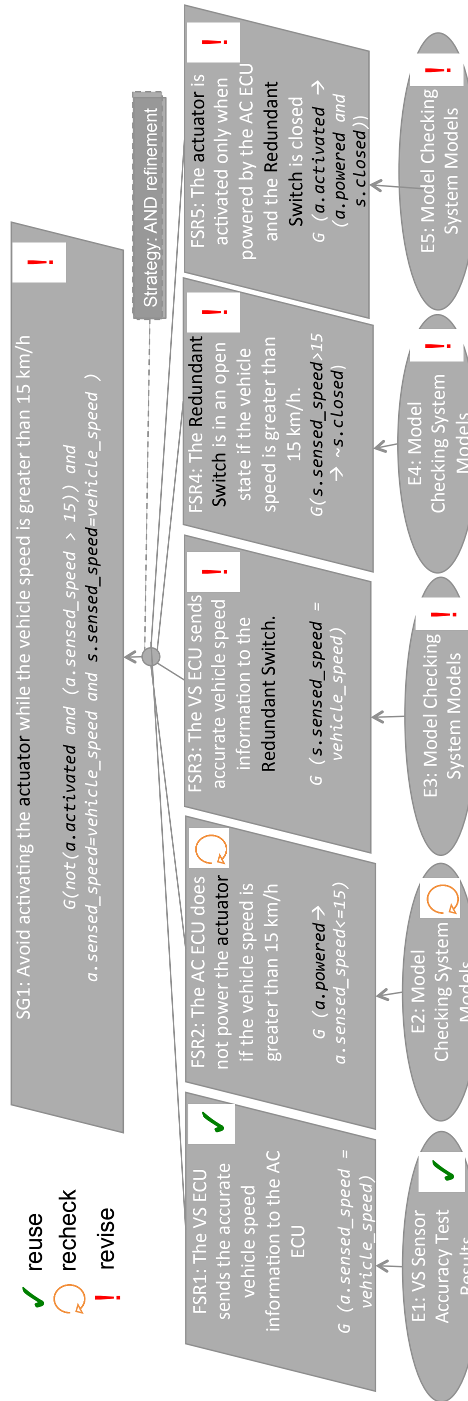


Figure 7.6: Goal tree after running the evolution algorithm.

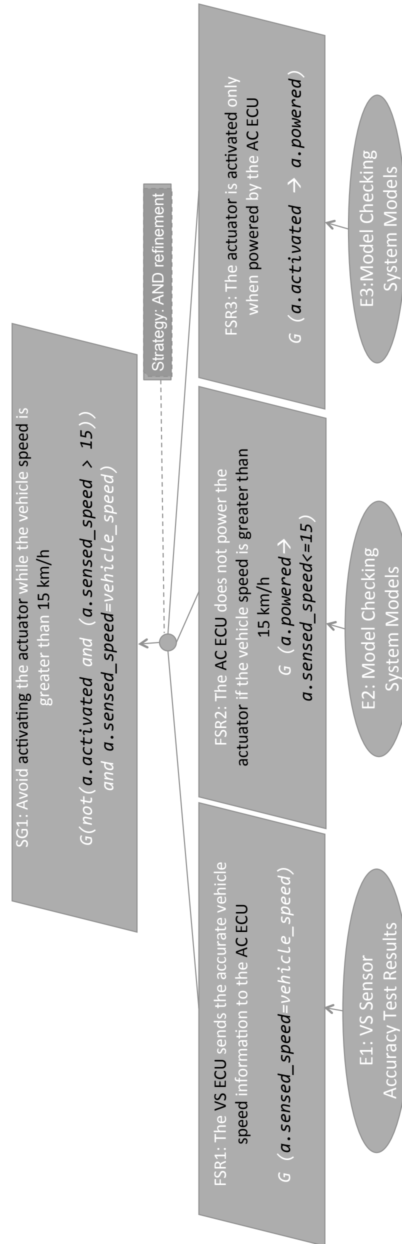


Figure 7.7: Final goal tree for power sliding door system without redundancy.

Chapter 8

Instantiating the Approach for Safety, Automotive and GSN

In the previous chapter, we introduced a generic model-based assurance case change impact assessment approach, that, while sound, was not particularly precise. In this chapter, we show how exploiting knowledge about system changes, the particular safety case language, and the standard can increase the precision of the impact assessment, reducing any unnecessary revision work required by a safety engineer. We present six precision improvement techniques illustrated on a GSN safety case used with ISO 26262.

The content of this chapter has been published in [Kokaly *et al.*(2017)].

8.1 Introduction

In this chapter, we build on our work in the previous chapter where we presented a model-based approach to perform impact assessment on an assurance case due to system changes. Our technique is applicable to assurance cases in general and ensures soundness, i.e., it does not miss any elements that are impacted. Yet, the approach is conservative. i.e., it can flag elements as impacted when they are not, resulting in “false positives”. Using knowledge about the system models, the safety case language and the standard under consideration, the precision of our approach can be improved, thus reducing unnecessary effort by the safety engineer.

Contributions. The contributions of this chapter are as follows:

1. We provide a model-based approach for impact assessment instantiated from Chapter 7 to GSN safety cases used with ISO 26262.

2. We identify and describe six techniques for improving the precision of the impact assessment approach.

Organization. The rest of the chapter is organized as follows: Section 8.2 describes how our model-based approach is instantiated for GSN safety cases linked to ISO 26262. Section 8.3 presents the techniques that can be used to improve the precision of our model-based impact assessment approach. Section 8.4 discusses related work, and Section 8.5 summarizes the chapter.

8.2 GSN Safety Case Impact Assessment

In this section, we present our generic safety case impact assessment approach from Chapter 7 specifically instantiated for GSN Safety Cases [Kelly and Weaver(2004)]. First, we define the GSN metamodel and the result of the impact assessment algorithm. Then, we describe the algorithm, which we name GSN-IA (GSN Impact Assessment), and the supporting model transformations.

8.2.1 GSN and Annotation Models

Figure 8.1 gives a fragment of the GSN metamodel extended with state information. A Goal has a truth state and we assume that the truth state is two-valued truth (*true*, *false*) and that every goal represents a claim about the system for which the truth can be determined (e.g., claim expressed as a temporal logic statement). Thus, for the time being, we preclude more fine-grained measures of truth (e.g., degrees of confidence) and goals that have fuzzy truth conditions, and leave as future work. A Solution represents some kind of evidence about the system and has a validity state that indicates whether the evidence is applicable or it is "stale" and must be regenerated (e.g., old test results). A Strategy is used to decompose goals (conclusions) into subgoals (premises), and its validity state indicates whether the strategy is a valid one for connecting its premise goals to its conclusion. Finally, a Context element describes assumptions on the elements it connects to, and also has a validity state.

We consider two ways that a change to the system can impact the elements of the safety case: (1) revise – the *content* of the element may have to be revised because it referred to a system element that has changed and the semantics of the content may have changed, and (2) recheck – the *state* of the element must be rechecked because it may have changed. For example, the goal "The power sliding door opens when the function DriverSwitch.RequestDoorOpen() is invoked and the vehicle speed is not greater than 15km/h." (see the class diagram in Figure 2.2 must be revised if the function name is changed to CommandDoorOpen() since the goal now refers to an element that does not exist. However, if some aspect of the system that affects door opening functionality changes, then the goal must be rechecked because it may no longer hold. We assume that after a revision, a recheck must take place; thus, at most one of these impacts can apply to an element. If an element is not impacted by a system change we say that it can be reused and mark it as reuse.

The purpose of executing our impact assessment algorithm, GSN-IA, on a safety case is to determine the impact type for each safety case element and to "mark" the element accordingly. This marking is stored in a simple annotation model with the metamodel shown in Figure 8.2b. Thus, an

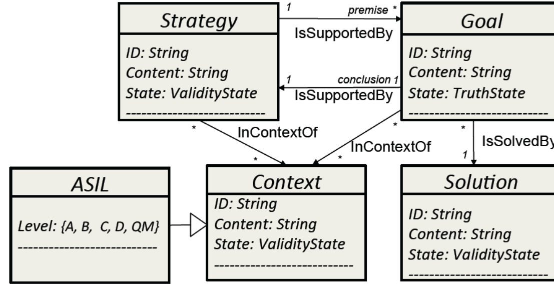
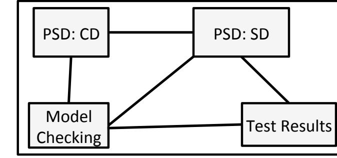
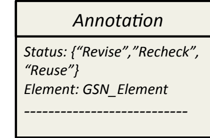


Figure 8.1: Fragment of GSN Metamodel extended with validity states.



(a) PSD System Megamodel



(b) Annotation Metamodel

Figure 8.2: PSD System Megamodel and Annotation Metamodel.

annotation model consists of an Annotation element for each GSN element that contains the marking as its Status attribute.

8.2.2 GSN-IA: GSN Impact Assessment Algorithm

Figure 8.3 shows the GSN-IA algorithm both in pseudocode and diagrammatically. The input to GSN-IA is the initial system model S and a safety case A connected by a traceability mapping R, the changed system S' and the delta D recording the changes between S and S'. Specifically, D is the triple $\langle C0a, C0d, C0m\rangle$ where $C0a$ is the set of elements added in S', $C0d$ is the set of elements deleted from S and $C0m$ is the set of modified elements that appear in both S and S'. These are shown in the top part of the diagram. GSN-IA is parameterized by the model slicer $\texttt{Slice}_{Sys}$ used to determine how change impact propagates within the system model – that is, we consider this slicer to be given as an input to GSN-IA. Note that our approach readily applies not only to singleton models but also to more realistic cases where the system is described by a heterogeneous collection of related models as a megamodel. We have defined a sound slicing approach for this case in Chapter 5. The output of GSN-IA is the model K that annotates A to indicate which elements are marked for revise, recheck or reuse.

GSN-IA uses several *model transformations* described below. In line 1, the Restrict transformation extracts the subset R'_A of traceability links from R that are also valid for S'. Lines 2 and 3 use the model slicer $\texttt{Slice}_{Sys}$ to expand the combined (using Union) set of changed elements in S and S', respectively, to all elements *potentially impacted* by the change. Then, in line 4, these potentially impacted elements are traced to A across the traceability relationships using the Trace transformation and combined to identify the subset of elements in A that must be rechecked. The subset of safety case elements for revision is identified in line 5 by tracing the deleted and modified elements of S to A. Note that the elements of A marked revise is a subset of those marked recheck. Only those that are directly traceable to changed elements of S may require revision; others only need to be

rechecked. In lines 6 and 7, the appropriate GSN slicer $\texttt{Slice}_{GSN_V}$ ($\texttt{Slice}_{GSN_R}$) is invoked to propagate each of the revise (recheck) subsets to dependent elements in A which are added to the recheck subset. Finally, line 8 invokes CreateAnnotation to construct the annotation model K from the identified subsets of A. The elements of the subset $C2_{\text{revise}}$ are marked revise; the remaining elements in the subset $C3_{\text{recheck2}}$ are marked recheck, and all other elements are marked reuse.

Algorithm: GSN-IA
Params: $\langle \texttt{Slice}_{Sys} \rangle$
Input: initial system model $S : Sys$, safety case $A : GSN$,
traceability map R, changed system megamodel $S' : Sys$,
delta $D = \langle C0a, C0d, C0m \rangle$
Output: Annotation K
1: $R'_A \leftarrow \texttt{Restrict}(R, D)$
2: $C1dm \leftarrow \texttt{Slice}_{Sys}(S, \texttt{Union}(C0d, C0m))$
3: $C1am \leftarrow \texttt{Slice}_{Sys}(S', \texttt{Union}(C0a, C0m))$
4: $C2_{\text{recheck}} \leftarrow \texttt{Union}(\texttt{Trace}(R, C1dm), \texttt{Trace}(R'_A, C1am))$
5: $C2_{\text{revise}} \leftarrow \texttt{Trace}(R, C0d)$
6: $C3_{\text{recheck1}} \leftarrow \texttt{Slice}_{GSN_V}(A, C2_{\text{revise}})$
7: $C3_{\text{recheck2}} \leftarrow \texttt{Slice}_{GSN_R}(A, \texttt{Union}(C2_{\text{recheck}}, C3_{\text{recheck1}}))$
8: $K \leftarrow \texttt{CreateAnnotation}(A, C3_{\text{recheck2}}, C2_{\text{revise}})$
9: **return** K

Figure 8.3: Algorithm for assessing impact of system changes on a GSN safety case.

Rule	**Element**	**Dependent Element(s)**
GSN_1	Goal G	1. All goals/strategies linked to G on either end of the IsSupportedBy relation. 2. All solutions linked to G via the IsSolvedBy relation.
GSN_2	Strategy S	All goals linked to S on either end of the IsSupportedBy relation.
GSN_3	Context C	1. All goals, strategies and solutions A that introduce C as the context via the InContextOf relation. 2. All goals, strategies and solutions that inherit C as the context (i.e., all children of A).
GSN_4	Solution S	All goals related to S via the IsSolvedBy relation.

Table 8.1: $\texttt{Slice}_{GSN_V}$ dependency rules.

Our $\texttt{Slice}_{GSN_V}$ slicer uses the dependency rules in Table 8.1 adapted from the set of propagation rules described in [Kelly and McDermid(2001b)] to identify elements to be marked for rechecking. For example, $\text{GSN}_{1.1}$ says that all goals and strategies linked to a goal G on either end of the IsSupportedBy relation are dependent on G (and are therefore marked "recheck"), if G is marked for revision. On the other hand, $\texttt{Slice}_{GSN_R}$ only uses two dependency rules to identify elements to be

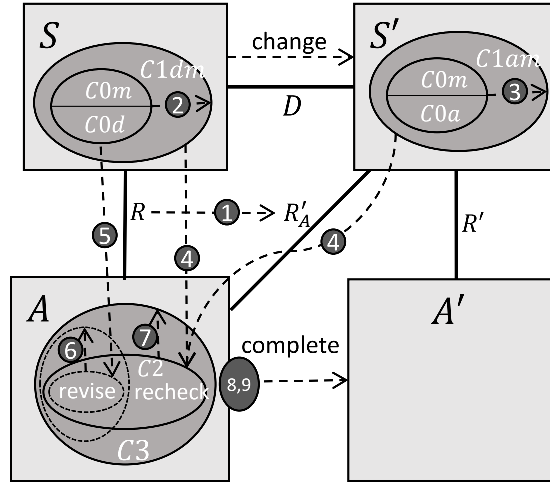


Figure 8.4: Visualization of GSN-IA algorithm.

marked for rechecking: (1) Conclusion goals depend on premise goals they are indirectly linked to by the same strategy, and (2) Goals depend on solutions they are linked to by the IsSolvedBy relation.

While $\texttt{Slice}_{GSN_V}$ only performs a one-step slice to find the revised elements' direct dependencies, $\texttt{Slice}_{GSN_R}$ works by continuously expanding a subset of elements in a GSN model to include its dependent elements until no further expansion is possible.

8.2.3 Illustration: PSD Example

In our PSD example, the change in the system is the removal of the redundant switch, so the delta D is $\langle\emptyset, (\text{RedundantSwitch}), \emptyset\rangle$. The change directly affects goals B3-6 shown in Figure 8.5, which refer to the Redundant Switch, and are therefore marked as revise by GSN-IA. The change also affects solutions SN3-6 which would include information about the Redundant Switch. Goal B2 refers to the AC ECU which is traced to the Redundant Switch in the PSD Class Diagram. $\texttt{Slice}_{Sys}$ would have detected that; therefore, B2 is marked recheck. Goal B1 does not link to any system components, so it does not appear in the result of $\texttt{Slice}_{Sys}$, and is therefore marked reuse. The remaining parts of the safety case elements are not traced directly to elements in the delta, but get marked using $\texttt{Slice}_{GSN_V}$ and $\texttt{Slice}_{GSN_R}$ described earlier. The result of GSN-IA is the annotation given on top of the original safety case and shown in Figure 8.5.

8.3 A More Precise Impact Assessment

The algorithm GSN-IA, presented in in Section 8.2, is conservative, i.e., more elements are marked recheck and revise than potentially necessary to still be sound. In this section, we present six different

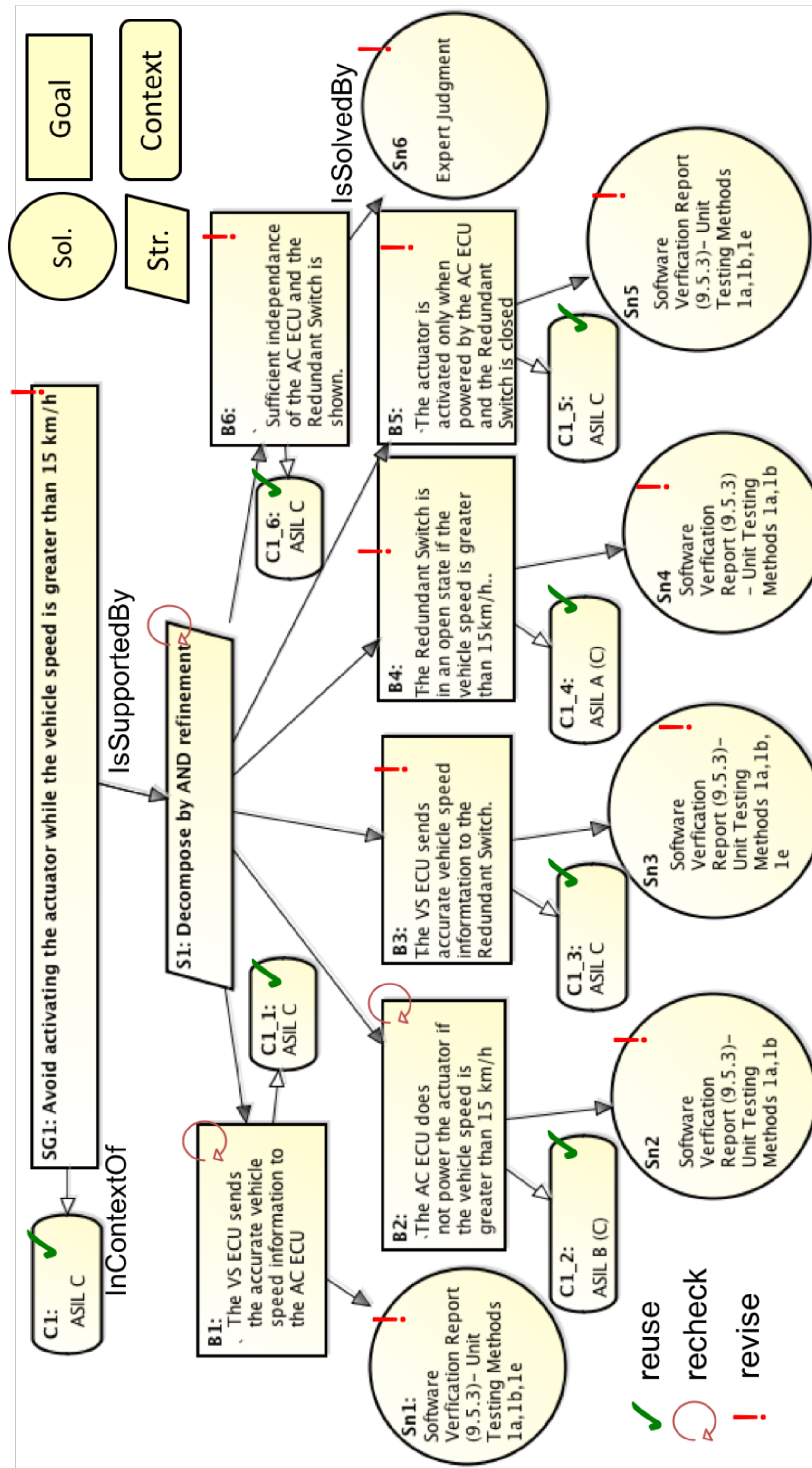


Figure 8.5: An annotated GSN safety case for PSD system after running GSN-IA.

$$
\begin{aligned}
Cost_{IA} &= Cost_{Revise} + Cost_{Recheck} \\
&= (Cost_{G_V} + Cost_{C_V} + Cost_{Sol_V} + Cost_{Str_V}) + (Cost_{G_R} + Cost_{C_R} + Cost_{Sol_R} + Cost_{Str_R}) \\
&= (\sum_{g \in G_V} K_V(1 + n(g)) + \sum_{c \in C_V} K_V(1 + n(c)) + \sum_{s \in Sol_V} K_V + \sum_{s \in Str_V} K_V) + \\
&(\sum_{g \in G_R} K_R + \sum_{c \in C_R} K_R + \sum_{s \in Sol_R} K_R + \sum_{s \in Str_R} K_R) \\
&= K_V(\sum_{g \in G_V}(1 + n(g)) + \sum_{c \in C_V}(1 + n(c)) + |Sol_V| + |Str_V|) + K_R(|G_R| + |C_R| + |Sol_R| + |Str_R|) \\
&= K_V(\sum_{g \in G_V}(1 + n(g)) + \sum_{c \in C_V}(1 + n(c)) + |Sol_V| + |Str_V|) + K_R(|E_R|), \text{ where:}
\end{aligned}
$$

- $Cost_{Revise}$ ($Cost_{Recheck}$): Cost of all revisions (rechecks).
- E_V (E_R): Number of total elements marked for revision (rechecking).
- G_V (G_R): Number of goals marked revise (recheck).
- C_V (C_R): Number of contexts marked revise (recheck).
- Str_V (Str_R): Number of strategies marked revise (recheck).
- Sol_V (Sol_R): Number of solutions marked for revise (recheck).
- $n(x)$: Number of identifiers in x marked for revise.
- K_V (K_R): Cost of performing a revision (a recheck).

Figure 8.6: Cost equation for effort incurred after an impact assessment.

techniques, T1-T6, aimed to improve the precision of GSN-IA. Together, they form a variant of GSN-IA, called GSN-IA-i (improved). The improvements in assigning annotation can be both at the level of safety case elements (goals, strategies, contexts and solutions), or finer, at the level of element identifiers. In order to validate GSN-IA-i, we use a metric $Cost_{IA}$ to compute the cost associated with revision and rechecking after impact assessment. The equation for $Cost_{IA}$ is shown in Figure 8.6. For each technique, we describe the current state of GSN-IA, show how to improve the precision in each case (GSN-IA + Ti), present the prerequisites to ensure its soundness, and illustrate it on the PSD example. The techniques are summarized in Table 8.2.

Technique	Improvement
1	$n(g) \downarrow, n(c) \downarrow$
2	$\lvert G_V \rvert \downarrow, \lvert C_V \rvert \downarrow$
3	$\lvert E_R \rvert \downarrow$
4	$\lvert E_R \rvert \downarrow$
5	$\lvert E_R \rvert \downarrow$
6	$\lvert E_R \rvert \downarrow$

Table 8.2: GSN-IA +Ti techniques and improvements.

8.3.1 T1: Increasing the Granularity of Traceability between the System and the Safety Case

GSN-IA: Trace links between the system and safety case provided to GSN-IA are assumed to link entire safety case elements to system elements. That is, if a change occurs in any of the linked system elements, the entire safety case element is marked for revision.

GSN-IA + T1: Trace links between the system and safety case connect *identifiers* in safety case elements to corresponding system elements. Annotations are then assigned to safety case element identifiers rather than to entire elements.

Improvement: With more fine-grained trace links, GSN-IA + T1 can identify which specific identifiers in a safety case element should be marked for revision, allowing the safety engineer to focus on revising only those parts instead of the entire element. This in turn decreases the number of unnecessary identifier revisions, i.e., $n(g)$ and $n(c)$, since only goals and context nodes are assumed to have identifiers traceable to the system, thus decreasing the overall cost.

Prerequisites: A safety case language that clearly distinguishes identifiers from other text, ensuring that the finer-grained trace links cover at least all the originally covered links in order to preserve soundness of the technique.

Example: In the PSD system, the goal B3 "The VS ECU sends accurate vehicle speed information to the Redundant Switch" can be traced to both VS ECU and Redundant Switch components. Currently, when either VS ECU or Redundant Switch changes, GSN-IA marks the entire goal revise. A more fine-grained traceability would link the identifier "VS ECU" to VS ECU in the system and the identifier "Redundant Switch" to the Redundant Switch in the system. Now, if Redundant Switch changes in the system but VS ECU does not, then only the identifier "Redundant Switch" in goal B3 needs to be marked for revision, while the rest of the goal can be reused.

Discussion: Traceability between the system and its safety case can be established at different levels of granularity. Formal safety case languages have clearly defined *identifiers*, thus they can easily be traced to the appropriate system elements. For example, the author of [Kelly(1997)] defines a six-step approach for creating well-formed GSN goal structures that in turn aid in a finer-grained system traceability. For languages that only use natural language to describe goals, this fine grained traceability may not be feasible.

8.3.2 T2: Identifying Sensitivity of Safety Case to System Changes

GSN-IA: Any change to a system element will cause its associated element in the safety case to be marked for revision.

GSN-IA + T2: We mark the safety case element for revision only if it is required by the type of system change.

Improvement: Unnecessary revisions of safety case element are minimized by identifying cases where a system change should actually impact the element, and where it can be ignored. This in turn decreases the number of goal ($|G_V|$) and context ($|C_V|$) elements marked for revision, decreasing the overall cost.

Prerequisites: For each model type in the system megamodel, a sensitivity table that lists all element types of that model and the kinds of changes that they can undergo, and, for each trace link between the system and the safety case, the type of change the link is sensitive to. We assume that the *types* of changes that occur as part of the system evolution are captured with each of the corresponding changes in the *Delta* we are provided. Since the assignment of sensitivity to change is performed by the domain expert, we require these assignments to be correct to ensure the preservation of soundness.

Example: In the PSD System, the class Door in the Class Diagram model has an attribute *state*, which is an enumeration with possible values *open* and *closed*. Assume a goal such as "If the door state is open and the speed is greater than 15km/h, the driver is notified.". Currently, if we add a new option to the door state (e.g., "stuck"), that is considered a change in the door state, which marks the goal for revision. However, such a change (an attribute enumeration extension) should not impact the goal which is only concerned with the door state being open. If we do not add that type of change in the sensitivity list of that particular trace link between system and goal, we are able to ignore it and allow the goal to be reused.

Discussion: In the example above, if the goal had been "If the door state is *not closed* and the speed is greater than 15km/h, the driver is notified.", then the change should have impacted this goal, as "stuck" is considered "not closed". We assume that goals are structured in a way that specific states are identified; if they are not, T2 cannot be used. Interestingly, in such a case, the goal would have to be marked revise, which may allow detecting missing test cases or other evidence for the "stuck" state.

8.3.3 T3: Understanding Semantics of Strategies

GST-IA: Any truth valuation change of the premise goals of a strategy lead to rechecking the conclusion goal.

GSN-IA + T3: Here, we use semantic knowledge, i.e., which changes in truth values of the premises do not affect the truth value of the conclusion.

Improvement: We limit the unnecessary propagation of recheck annotations across the safety case, thus $|E_R|$ decreases, causing the overall cost to decrease.

Prerequisites: Semantics of the strategies connecting premise and conclusion goals. This applies to a fixed set of known strategies and not to strategies expressed in natural language. Soundness is preserved since we are using sound semantics of logical connectives to make decisions.

Example: Assume in the PSD system that SG1 was connected to its subgoals B1-B6 via an "OR" decomposition strategy (as opposed to an "AND"). Also assume that currently all of B1-B6 have *true* states. This means that SG1 is also evaluated to *true*. If the system changes so that B5-B6 are marked recheck, we don't need to mark SG1 recheck since, due to disjunction, it must still be *true*.

Discussion: Marking a premise of an "OR" strategy recheck (while other premises are marked reuse) can impact the overall confidence in the argument, as the premise can become *false* after the recheck is performed. We do not take confidence into account at this point and consider it future work.

8.3.4 T4: Decoupling Revision from Rechecking

GSN-IA: Forces a recheck every time an element is marked revise.

GSN-IA + T4: By knowing circumstances under which revising a goal will not impact its truth value, we require a recheck after a revision only when necessary.

Improvement: Eliminating unnecessary rechecks after revisions leads to possibly decreasing $|E_R|$ and, therefore, the overall cost.

Prerequisites: An extra column in the sensitivity table described in T2 that lists if a particular type of change affects the truth value of a goal. We require correctness of assignments of *changes* to their *effect on goal truth values* as well as completeness of trace links to ensure soundness of the approach.

Example: In the PSD system, changing the name of a system element such that it does not conflict with other names (e.g., Redundant Switch is renamed to Extra Switch) will cause the goals referring to that element (e.g., goal B3) to be marked for revision. However, since changing the name does not impact the truth state of the goal, rechecking can be skipped. Other examples include capitalization of names, spelling corrections or language translations, such that the renaming is done consistently in both the system and the safety case.

8.3.5 T5: Strengthened Solutions do not Impact Associated Goals

GSN-IA: If a piece of evidence that a solution points to changes, the goal supported by that solution is always marked recheck.

GSN-IA + T5: A change to a solution that strengthens it should not affect its support for associated goals.

Improvement: Understanding which changes in solutions do not necessitate a rechecking of associated goals can reduce the unnecessary goal rechecks. Thus, $|E_R|$ decreases causing the overall cost to decrease.

Prerequisites: A sensitivity table (similar to T2) that identifies, for each type of evidence, the types of changes it can undergo, and for each "isSupportedBy" link between a solution node pointing to this kind of evidence and a goal, whether or not it is sensitive to each kind of change. Assignments of *changes* to their *effect on goal truth values* need to be correct to guarantee soundness.

Example: Assume that B1 was "The VS ECU sends accurate vehicle speed information to the AC ECU *90% of the time*" and that it was linked to a solution with test cases which showed accuracy 90% of the time. If the system changes so that the test cases can now demonstrate accuracy 100% of the time, this does not affect goal B1, meaning that it should not be marked for rechecking.

8.3.6 T6: Exploiting Knowledge about ASIL Work-Product Dependencies and ASIL Propagation and Decomposition Rules

GSN-IA: Does not take into account how changes in the system impact ASILs.

GSN-IA + T6: Determine how ASILs should change due to system changes by using knowledge about ASIL work-product dependencies and ASIL propagation and decomposition rules.

Improvement: Increase in precision due to distinguishing between changes to the goals and changes to the ASILs, potentially decreasing the number of required goal rechecks. This decreases $|E_R|$, thereby decreasing the overall cost.

Prerequisites: Dependency tables from ISO 26262 Part 6 that describe the types of methods for each work product required to achieve certain ASILs, and ASIL decomposition and propagation rules as presented in ISO 26262 (refer to Section 6.2). We assume the soundness of the tables and ASIL propagation and decomposition rules in order to guarantee soundness of our approach.

Example: We present two examples in the PSD system:

1. If method `1e` (Back-to-back comparison test between models and code) used for unit testing as part of the Software Verification Report work product for goal B1 is deleted, the ASIL for B1 supported by Sn1 changes from ASIL C to ASIL B based on the table in Figure 6.7. This would in turn impact the ASIL on SG1, since the ASIL propagation rule no longer holds. In this case, claims B1 and SG1 themselves are not impacted, only their ASIL levels are.
2. With redundancy present in the PSD system, ASIL decomposition was used to allocate

ASIL B to B2 and ASIL A to B4 (decomposed from ASIL C). B6 was added to demonstrate sufficient independence of the Redundant Switch element from the AC ECU as required by ASIL decomposition. When the system changes and the redundant switch is deleted, requirements B4 and B6 are marked for revision, causing the original decomposition rule to be impacted. B2 is only marked recheck, but its ASIL level will be marked for revision (from ASIL B to ASIL C) to respect ASIL propagation rules from SG1. The impact assessment now flags both C1_2 and Sn2 for revision. Ideally, the safety engineer will revise Sn2 to be strengthened (e.g., unit testing method 1e is added) to increase the ASIL on B2 to level C.

8.3.7 PSD Example Cost Comparison with T1

Assume that the revision cost K_V is 2 units and the rechecking cost K_R is 1 unit[1]. On the PSD example, GSN-IA produced an annotation with 8 elements marked revise (4 goals, 4 solutions) and 4 marked recheck. Goals marked revise have the following number of identifiers: B3 has 3 (VS ECU, vehicle speed, Redundant Switch), and similary, B4, B5 and B6 each respectively have 3, 6 and 2 identifiers. The cost incurred after GSN-IA is $2 \times ((1+3)+(1+3)+(1+6)+(1+2)+4)+1 \times (4) = 48$ units.

Using T1, for example, changes to the redundant switch link only to the *Redundant Switch* identifier in goals B3-B6 (as opposed to the entire goals), dropping the number of revised elements in each of these goals to only 1 (as opposed to marking all the identifiers in the goal for revision). The cost after running GSN-IA + T1 is $2 \times ((1+1)+(1+1)+(1+1)+(1+1)+4)+1 \times (4) = 28$ units, representing a clear improvement. Other techniques can be assessed in a similar manner.

8.4 Related Work

Model-based Approaches to Safety Case Management. Many methods for modeling safety cases have been proposed, including goal models and requirements models [Ghanavati *et al.*(2011), Brunel and Cazin(2012)] and GSN [Kelly and Weaver(2004)]. The latter is arguably the most widely used model-based approach to improving the structure safety arguments. Building on GSN, Habli et. al. [Habli *et al.*(2010)] examine how model-driven development can provide a basis for the systematic generation of functional safety requirements and demonstrates how an automotive safety case can be developed. Gallina [Gallina(2014)] proposes a model-driven safety certification method to derive arguments as goal structures given in GSN from process models. The process is illustrated by generating arguments in the context of ISO 26262. We consider this category of work complimentary to ours; we do not focus on safety case construction but instead assume presence of a safety case and focus on assessing the impact of system changes on it.

Safety Case Maintenance. Kelly [Kelly and McDermid(2001b)] presents a tool-supported process, based on GSN, that facilitates a systematic safety case impact assessment. The work by Li

[1] In practice, $K_V > K_R$, since revision requires more effort than rechecking.

et. al. [Li(2016)] proposes an assessment process to specify typical steps in the safety case assessment. The authors develop a graphical safety case editor for assessing GSN-based safety case and use the Evidential Reasoning (ER) algorithm to assess the overall confidence in a safety case. Jaradat and Bate [Jaradat and Bate(2016)] present two techniques that use safety contracts to facilitate maintenance of safety cases. As far as we are aware, none of the approaches provide a structured model-based algorithm for impact assessment, or consider methods for improving its efficiency. In the context of safety case maintenance, Bandur and McDermid [Bandur and McDermid(2015)] present a formalization of a logical subset of GSN with the aim of revealing the conditions which must be true in order to guarantee the internal consistency of a safety argument. This provides a sound basis for understanding logical relationships between components of a safety case and thus to enhance impact assessment.

8.5 Chapter Summary

In this chapter, we showed how using various sources of knowledge about the system changes, the particular safety case language and the safety standard can increase the precision of the previously presented impact assessment technique in Chapter 7, thus reducing the work required by the safety engineer. We presented six precision improvement techniques and illustrated our ideas using a GSN safety case used with ISO 26262.

Part IV

Tool Support & Validation

Chapter 9

Tool Support: *MMINT-A*

In this chapter, we present a tool *MMINT-A* that can, in the context of model-driven development, assess the impact of system changes on their assurance cases. To achieve this, *MMINT-A* implements the impact assessment algorithm from Chapter 8, and incorporates a graphical assurance case editor, an annotation mechanism, and summary tables for the assessment results. We demonstrate the usage of *MMINT-A* on the PSD system we have considered in this thesis.

The content of this chapter was published in [Fung *et al.*(2018)], where my contribution focused on guiding the structure and content of the paper, including creation of the example presented. The specifications for the tool came from my own work and ideas, and the development was done by the first author. I participated in the testing and evaluation of the tool.

9.1 Introduction

To facilitate the assurance case change impact assessment process, we have developed a collection of extensions to the *MMINT* model management framework presented in Chapter 3 to support automated Change Impact Assessment (CIA) on ACs, focusing specifically on the automotive domain. The resulting tool, *MMINT-A* (which is available at `https://github.com/nlsfung/MMINT`), identifies how different parts of an AC may be impacted by some given changes to the associated system, whether they can be reused in the updated AC or must be revised or rechecked for either state or content validity. Thus, the engineer can direct her efforts to reviewing and updating the appropriate parts of the AC.

Contributions and Organization. The main contribution of this chapter is the *MMINT-A* tool along with a description of its features. Section 9.2 lists the main requirements for the tool. We describe the extensions on top of *MMINT* to create *MMINT-A* in Section 9.3, and demonstrate its features using the PSD example in Section 9.4. We discuss related work in Section 9.5 and conclude in Section 9.6.

9.2 *MMINT-A* Requirements

The main high-level requirements for *MMINT-A* are:

1. The tool shall support an assurance case metamodel as defined in Chapter 8.
2. The tool shall provide an assurance case editor for creating ACs conforming to the defined metamodel and the GSN concrete syntax in [GSN(2011)].
3. The tool shall provide the user with a way to trace assurance case elements to system models (which appear as part of a megamodel in *MMINT*).
4. The tool shall support the notions of ASIL and ASIL decomposition from the automotive domain and model them as part of the AC metamodel, allowing the user to define ASILs in the editor provided.
5. The tool shall provide the user with a way to specify modified, deleted and added elements in the system models.
6. The tool shall implement the AC impact assessment workflow defined in Chapter 8.
7. The tool shall support syntactic checks that are not enforced automatically by the AC metamodel but are nevertheless required for well-formed ACs.
8. The tool shall provide the user a way to set the content validity of each node in the AC.
9. The tool shall present visualization of the impacted AC elements via revise, recheck, and reuse annotations.
10. The tool shall report on percentages of elements marked by each annotation.
11. The tool shall provide the user with information on backward traceability from annotated AC elements to the originating elements causing the impact from the system models.
12. The tool shall provide the user with a report on the cost of performing the impact assessment based on the cost formula defined in Chapter 8.

9.3 Extensions for *MMINT-A*

Recall from Chapter 3 that *MMINT* is built on top of the Eclipse platform and is designed for managing collections of related models (i.e., megamodels) that are represented as MIDs (Model Interconnection Diagrams). In particular, *MMINT* supports both the "instance" level, in which models, megamodels and their relations are instantiated and manipulated, as well as the "type" level, in which the necessary metamodels, relation types and model operators are defined. For example, a metamodel for UML class diagrams can be created on top of the Eclipse platform and plugged

into *MMINT* via the type support runtime layer. The Type MID would then be populated with metadata about the new metamodel, allowing UML class diagrams to be created and manipulated inside megamodels using the MID editor and the ModelRel (model relation) editor. With the workflow editor, new operators can be created by connecting pre-defined model operators into a directed, acyclic network, with the roots being the input models to the workflow and the leaves being the outputs.

MMINT-A is comprised of a collection of extensions to *MMINT* for instantiating and operating on ACs. However, since the impact assessment is driven by changes to the system, *MMINT-A* also requires the availability of appropriate metamodels (and editors) for instantiating the desired system models. Availability of operators, such as slicers, for CIA on individual system models is also assumed since they form part of the overall procedure executed by *MMINT-A*. Currently, *MMINT* supports CIA on simplified versions of UML class diagrams (CD), sequence diagrams (SD) and state machines (SM), which involved: (1) creating the metamodels using EMF (Eclipse Modeling Framework) [Eclipse(2018a)], (2) creating the corresponding editors using Sirius [Eclipse(2018b)], (3) implementing each slicer as a Java class, and (4) incorporating them into *MMINT* by editing their plug-in files.

9.3.1 Assurance Case Metamodel

Our metamodel for ACs (Figure 9.1) is derived from the Goal Structuring Notation (GSN) version 1 [Attwood *et al.*(2011)] in which an AC is modelled as a directed acyclic network of six types of elements: goals, strategies, solutions, contexts, justifications and assumptions. The former three form the core of an AC and are connected to each other via "supported-by" relations, with a top level goal as the root and the solutions as leaves, while the latter three are connected to the core via "in-context-of" relations. Each of these elements can also be given a unique identifier and description in accordance to the standard, but we extended it by adding states to goals and solutions which, although unnecessary for CIA in *MMINT-A*, allow the user to indicate, respectively, their truth values and the currentness of their evidence as part of the overall change management process.

Furthermore, to capture the CIA results, we introduced annotations to our metamodel, which we previously modelled in Chapter 8 as comprising three types: *Reuse*, to indicate that the element is not impacted; *Recheck*, to indicate that the element may no longer be valid and needs a recheck; and *Revise*, to indicate that the element's contents require changing. However, in *MMINT-A*, we also distinguish between: (1) recheck *content*, which indicates that the element's content may (but not necessarily) require revision, and (2) recheck *state*, which is applicable to goals and solutions only and indicates that the element's content, while reusable, may no longer be supported by the underlying sub-goals or evidence.

Focusing on the automotive domain, certain domain-specific features were also incorporated into our metamodel that can be disregarded in general. Specifically, goals are modelled to contain an optional ASIL (automotive safety integrity level) attribute that captures the inherent safety risk of

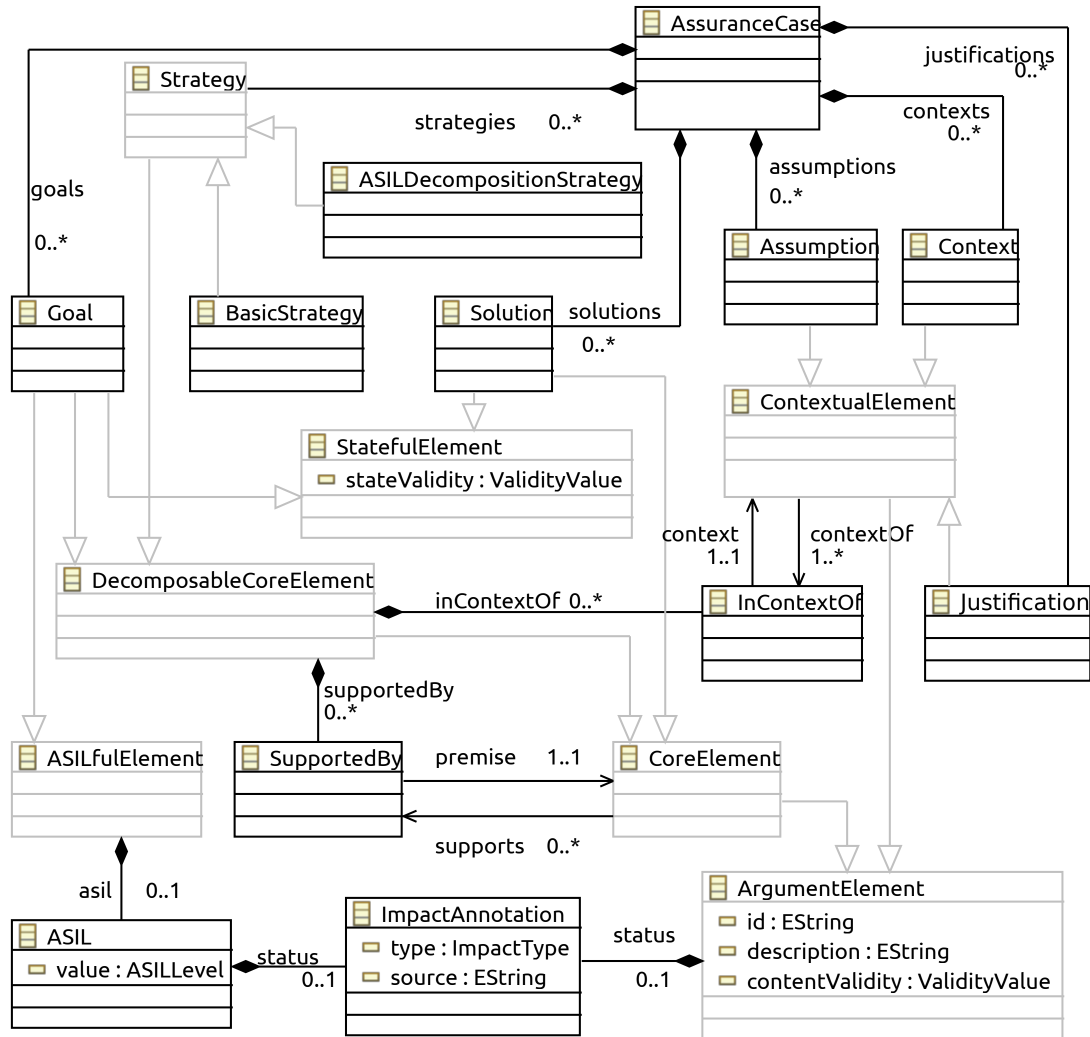


Figure 9.1: The AC metamodel in *MMINT-A*. Concrete and abstract classes are distinguished with black and grey borders, respectively.

the associated system component and is annotated separately from its goal for CIA. ASILs of sub-goals are generally inherited from their parent goals, but in accordance to ISO 26262 [ISO(2011)], they can be decomposed following certain conditions, which are captured in *MMINT-A* using a sub-type of `Strategy`, namely, `ASILDecompositionStrategy`.

9.3.2 Assurance Case Editor

Figure 9.2 shows a screenshot of the editor that was implemented on top of Sirius and comprises multiple "views" for creating and visualizing ACs. In the main view (left), ACs are visualized

Figure 9.2: Screenshot of the AC editor in *MMINT-A*. The main graphical view is on the left, the statistics table upper right, and the impact trace table lower right.

in accordance to the GSN standard but with additional decorations for ASILs and annotations. In particular, ASILs are represented as small, rectangular nodes bordering the goal nodes, while annotations are represented as exclamation marks, circular arrows and check marks to denote Revise, Recheck and Reuse, respectively. The subscripts "C" and "S" are used to disambiguate Recheck Content and Recheck State.

Although compliant with the GSN standard, this graphical representation may not always be the most appropriate. The user may wish to, for example, quickly analyze the amount of impact different changes have on an assurance case or review the source of each annotation. Therefore, to address these use cases, we created two tabular representations to summarize the results of the CIA:

- **Annotation report support.** This helps address the question: *How many elements are marked revise/recheck/reuse in the safety case following a particular change (i.e., what is the overall impact of a change on the reusability of a safety case)?* The resulting table is shown in the upper right in Figure 9.2. It displays the number (and percentage) of each type of node that are annotated for revision, rechecking or reuse.
- **Backward traceability support.** This helps address the question: *What system elements caused a safety case element to be marked "revise"?* The resulting table is shown in the lower right in Figure 9.2. It displays the type of impact for each node and the source of the impact. For example, Figure 9.2 shows that goal `G1.2` must be revised because of a change in the class `Redundant Switch`.

9.3.3 Assurance Case Slicers

As part of our overall AC CIA, we make use of two AC slicers based on the GSN slicers presented in Chapter 8. The *Revise Slicer* uses rules V_1 to V_4 in Table 9.1 to identify elements to be rechecked given an element marked for revision (which applies to the content and not the state of an element). The *Recheck Slicer* uses the rules C_1 and C_2 in Table 9.1 to identify elements to be rechecked given another element marked for rechecking. Note that the Recheck Slicer applies only to state rechecks and that while the Revise Slicer only performs a one-step slice to find direct dependencies of the revised elements, the Recheck Slicer recursively expands a subset of AC elements to include its dependent elements until closure is reached. The main changes from the GSN slicing rules presented in Chapter 8 are the following:

1. The updated rules differentiate between *content* and *state*. *Content* applies to all AC elements and refers to the text, while *state* applies only to goals (representing a truth value) and solutions (indicates whether the evidence is up-to-date or not).
2. The updated rules differentiate between *revise* (on content only) and *recheck* (on content or state).

Rule	Element	Dependent Element(s) (Annotation)
V_1	Goal G	All strategies linked to G on either end of the IsSupportedBy relation (recheck content).
V_2	Strategy S	1. All goals that S supports (recheck state). 2. All goals or solutions that support S (recheck content). 3. All justifications that are used to justify S (recheck content).
V_3	Context C	1. All goals, strategies and solutions A that introduce C as the context via the InContextOf relation (recheck content). 2. All goals, strategies and solutions that inherit C as the context (i.e., all children of A) (recheck content).
V_4	Solution S	All strategies that S supports (recheck content).
C_1	Goal G	All parent goals that are linked to G by the same strategy (recheck content).
C_2	Solution S	All goals that are linked to S by the same strategy (recheck content).

Table 9.1: The assurance case slicer dependency rules.

Additionally, since we consider ASIL as its own element in the AC, the updated rules support ASIL-aware impact assessment (e.g., by removing a redundancy mechanism, an ASIL decomposition strategy is no longer valid, and ASILs of the corresponding goals must also be rechecked). The rule introduced is the following: If a subgoal of an ASIL decomposition strategy is marked for *revision*, we mark its ASIL for *recheck content*, the linked strategy for *recheck content* and the independence goal for *recheck content* (unless it is marked for *revise content* due to direct traceability with the system). To implement this in the tool, we introduce a special type of strategy (ASIL decomposition strategy) which has two sub-goals and an independence goal as seen in Figure 9.1.

9.3.4 Change Impact Assessment Algorithm

MMINT-A implements our AC CIA algorithm from Chapter 8 which accepts as inputs the original and the updated system megamodel, the set of changes made (i.e., additions, deletions and modifications) as well as the AC and its relation to the megamodel. However, the *MMINT-A* implementation assumes that additions can only impact other model elements indirectly via the modifications and deletions required to accommodate them. Thus, the implemented algorithm (see Figure 9.4) does not require the added model elements nor the updated system megamodel.

This algorithm is encoded as a *MMINT* workflow with 13 model operators as seen in Figure 9.3. To ensure that the workflow is independent of any specific model types for the input system megamodel, it utilizes higher-level collection-based operators (particularly, **map** from Chapter 4), enabling it to apply the appropriate operators to the appropriate models in the system megamodel at runtime.

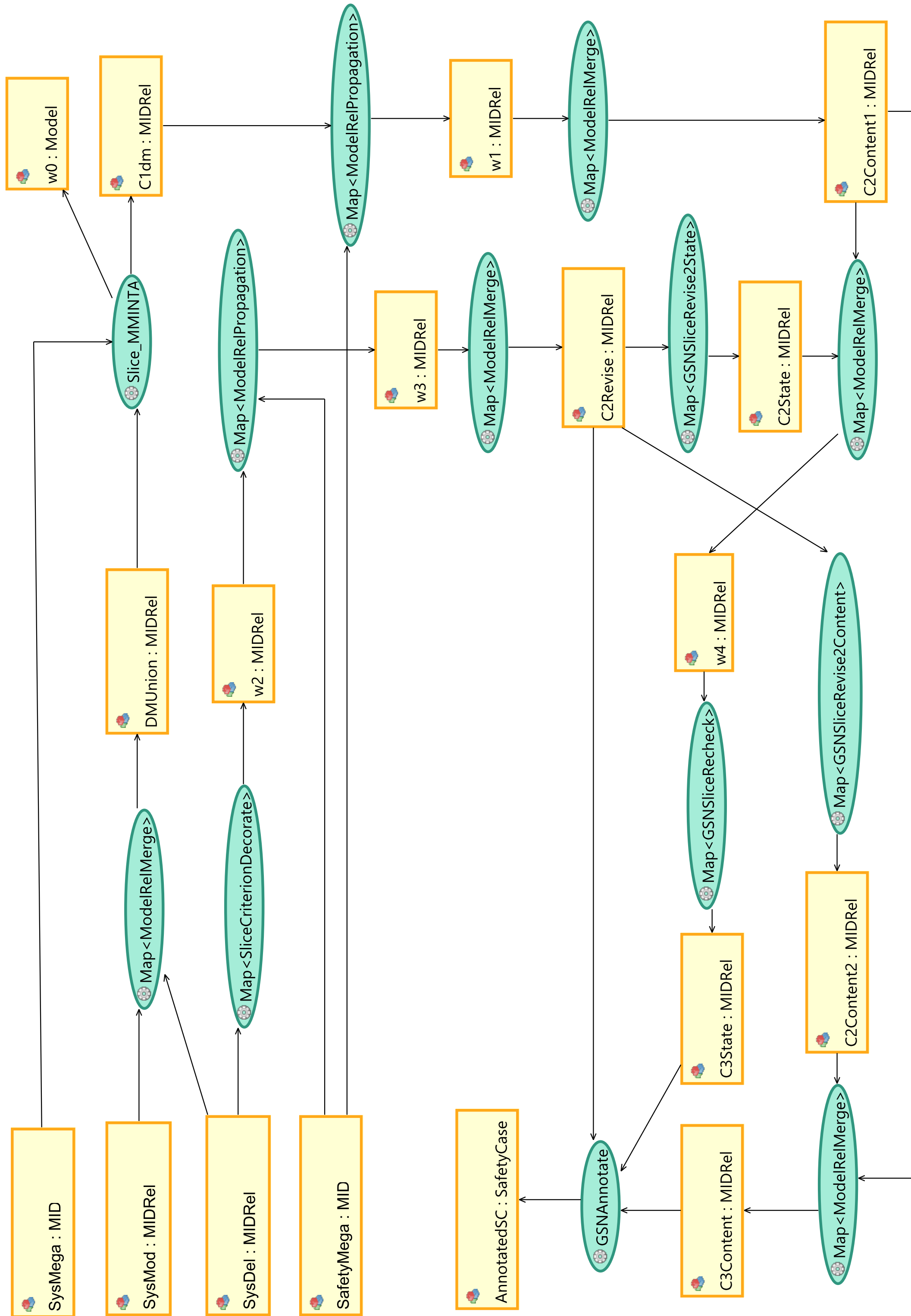
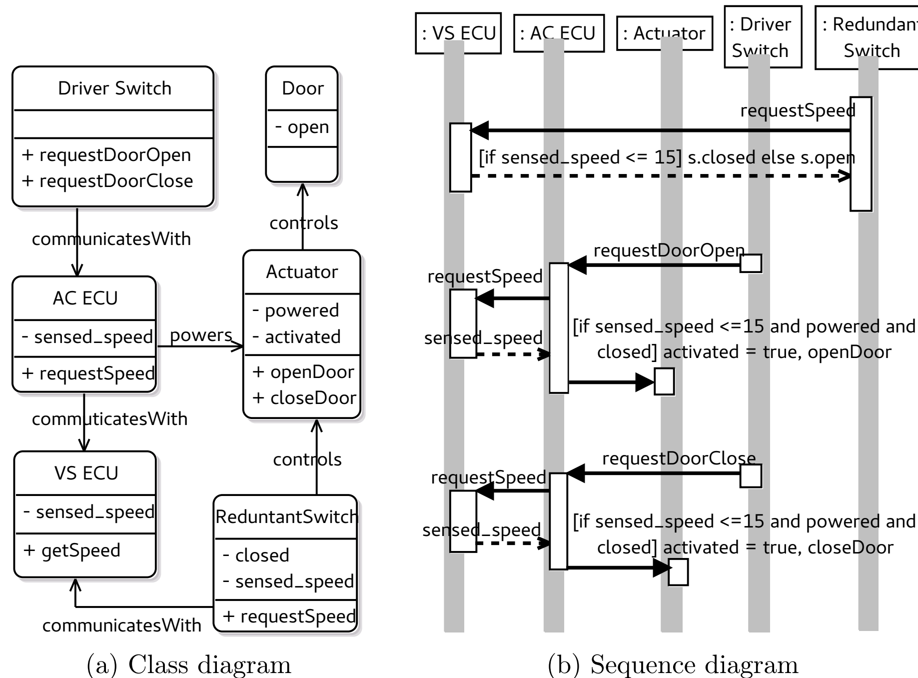


Figure 9.3: The assurance case CIA workflow in *MMINT*

1. Perform CIA on the system megamodel itself.
2. Propagate results to the AC to obtain the elements requiring content recheck.
3. Identify AC elements requiring revision from the deleted system model elements.
4. Apply Revise Slicer on results of step 3.
5. Merge and apply Recheck Slicer on results of steps 2 and 4.
6. Annotate the AC. The results of steps 2, 3 and 5 are marked for content recheck, revision, and state recheck, respectively.

Figure 9.4: *MMINT-A* impact assessment algorithm.



(a) Class diagram (b) Sequence diagram

Figure 9.5: Models for the PSD system in *MMINT-A*

9.4 Power Sliding Door Example

As part of our evaluation process, we used *MMINT-A* on the PSD running example. The system is modelled using a class diagram (CD) and a sequence diagram (SD) as shown in Figure 9.5, and it is associated with an AC comprising 22 nodes as shown later in Figure 9.6.

The overall goal (`G1`) for system safety is decomposed into four main subgoals (`G1.1` to `G1.4`), each of which is decomposed further until they are directly supported by the appropriate evidence. The third subgoal (`G1.3`) illustrates ASIL decomposition, i.e., how introducing an independent

redundant switch to the system allows a goal with a high ASIL (C) to be satisfied by subgoals with a lower ASIL in accordance to ISO 26262.

For the case study, we suppose that the redundant switch is removed, and we wish to analyze its potential impact on the AC. To achieve this using *MMINT-A*, we created a megamodel for the PSD by incorporating the CD and SD models into a MID and adding a relation between them using the ModelRel editor; the appropriate trace links to include in the relation were determined by matching the names of the elements in the class and sequence diagrams. Relations were created similarly between the AC and the system megamodel, all of which formed the inputs to the CIA algorithm, with the original change being the deletion of the redundant switch class and all of its attributes and operations.

The results of executing the workflow in *MMINT-A* (see Figure 9.6) agree with the manual results presented in Chapter 8. For example, Figure 9.6 shows that all AC elements that refer directly to the redundant switch must be revised, while any related elements must be rechecked for their content (and/or state) validity. Also, by removing the redundancy mechanism, the ASIL decomposition strategy is no longer valid, thus the ASILs of the corresponding goals must also be revised.

9.5 Related Work

As we showed in the survey presented in Section 6.4, a multitude of tools have been developed over the past two decades to support various aspects of working with ACs, many of which can perform CIA like *MMINT-A*. For example, D-MILS [University of York(2015)] and AutoFocus 3 (ExplicitCase) [Cârlan *et al.*(2017)] enable CIA by supporting trace links between system artefacts and the system AC, but unlike *MMINT-A*, they do not employ slicers to detect indirect impacts of change. Other tools, such as ENTRUST [Calinescu *et al.*(2017)] and Evidential Tool Bus (ETB) [Cruanes *et al.*(2013)], remove the need for CIA altogether by automatically propagating changes in the system artefacts to the AC. However, ENTRUST only supports certain changes to the AC, while ECB can only propagate changes to the underlying evidence.

Although the CIA functionality proposed and implemented in *MMINT-A* can be implemented on top of existing AC tools, these tools are generally highly specialized, making it impractical to adapt them for the automotive domain. On the other hand, before *MMINT-A* can become a usable tool itself, it must be extended with many "standard" features such as strong support for AC creation and assessment. In fact, unlike D-MILS and AutoFocus 3, *MMINT-A* can only support trace links to system models; other system artefacts such as natural language documents are not yet incorporated into the tool.

9.6 Chapter Summary

In this chapter, we presented the features of *MMINT-A* and demonstrated how it supports the maintenance of ACs, specifically by performing CIA on them. However, since *MMINT-A* is built

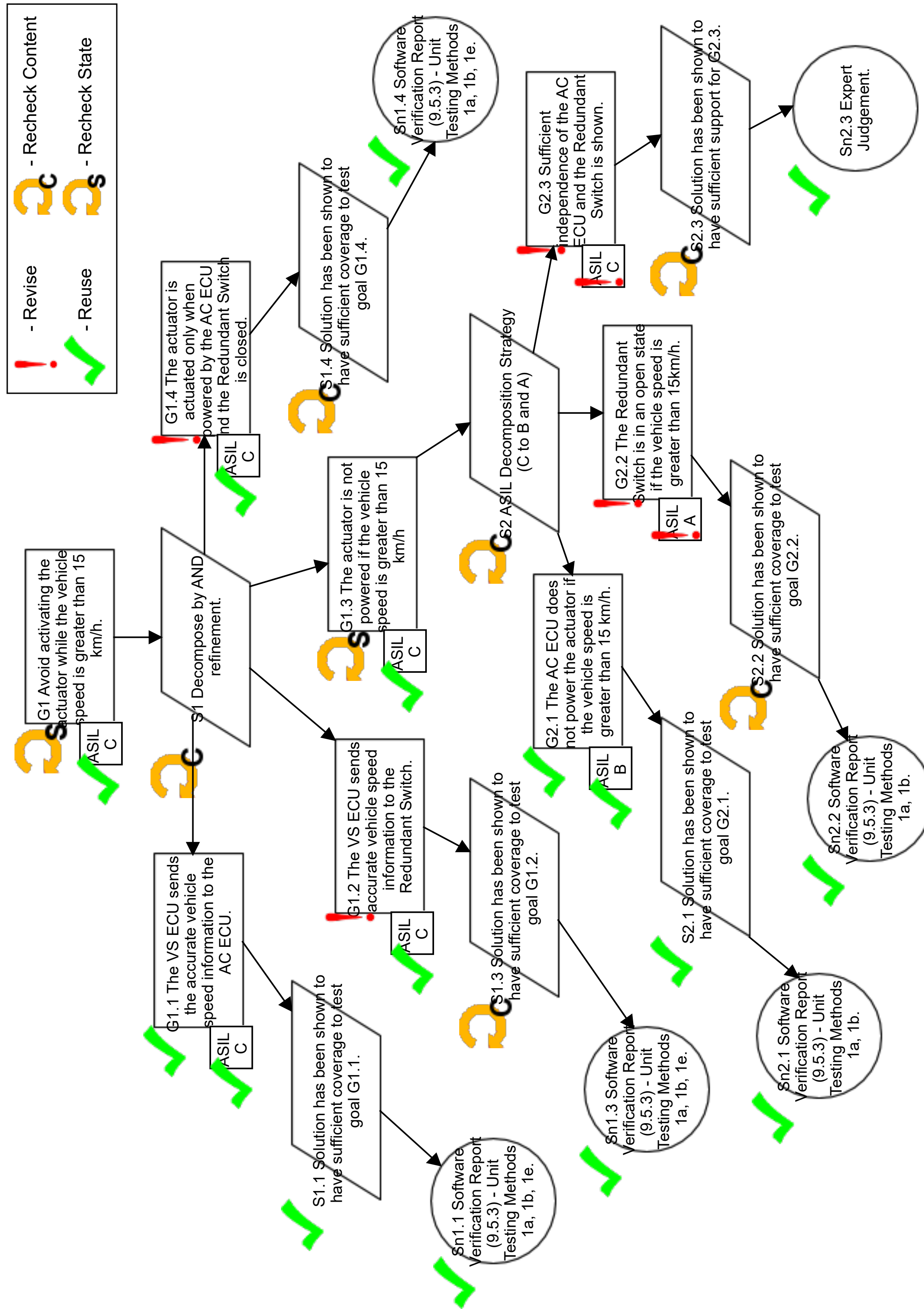


Figure 9.6: The PSD AC after change impact assessment in *MMINT-A*.

on top of the generic *MMINT* model management framework, it can be easily extended beyond CIA to address other scenarios, including those presented in Chapter 1 for regulatory compliance management. In fact, with the appropriate changes to the AC metamodel, *MMINT-A* can also be applied to non-automotive domains, but because of the focus on models, *MMINT-A* (and *MMINT*) are best suited for model-based software systems.

Chapter 10

Case Study: Lane Management System

In this chapter, we present the use of *MMINT-A* on a Lane Management System (LMS) from the automotive domain. We start by explaining the goal of the case study in Section 10.1. We describe the LMS system in Section 10.2 and present a safety case we have constructed for it in Section 10.3. We demonstrate the use of the tool on various change scenarios over LMS in Section 10.4. We conclude with a summary in Section 10.5.

The LMS specification and models in this chapter came from the literature [Blazy *et al.*(2014)], the safety case, the change scenarios and results are contributions of this thesis.

10.1 Introduction

In Chapter 9, we demonstrated the use of *MMINT-A* on the running PSD example. In this chapter, we apply our approach and *MMINT-A* on a larger system, namely, the Lane Management System (LMS) from the automotive domain. LMS is comprised of one class diagram, four sequence diagrams and four state machines (a total of nine system design models), with a 100-node assurance case (we consider ASILs their own nodes). We will discuss LMS in more detail in Section 10.2.

Recall that the goal of performing CIA on an assurance case, given a change in the system, is to compute a set of (potentially) impacted safety case elements that includes: *all* the *actually impacted* elements (**high recall**) and *very few* of the *non-impacted* elements (**high precision**), thus decreasing the cost associated with the effort of performing a manual impact assessment.

The research question that we wish to address in this case study is: Given different kinds of system design changes on the LMS system, does the application of our assurance case impact assessment approach indeed reduce the effort/cost of performing CIA compared to a manual approach?

To do so, we use the cost formula derived in Chapter 8 and shown in Figure 8.6, and we make the following modifications and assumptions for simplification:

1. We do not distinguish between the cost of rechecking content and rechecking state of a node.
2. We assume for the sake of computation that $K_V = 2$ and $K_R = 1$ (in practice, $K_V > K_R$, since revision requires more effort than rechecking).
3. Since we consider ASIL nodes separately in *MMINT-A*, we add to the formula $Cost_{AV}$ (ASIL revise) and $Cost_{AR}$ (ASIL recheck) with K_V (K_R) being the cost of performing an ASIL revise (recheck), and $A_V(A_R)$ being the number of ASIL nodes marked revise (recheck). We include each of A_V and A_R in E_V and E_R, respectively, absorbing the ASIL annotations in the total number of overall element annotations.
4. We assume that a fine-grained traceability (i.e., traceability to individual identifiers in a GSN element rather than to the whole element) is not available, and therefore $n(x) = 0$ for all x.

With the above applied to the formula, we end up with the following simplified version of the cost equation, which we will use in the scenarios considered in this case study in Section 10.4.

$$Cost_{CIA} = K_V \times E_V + K_R \times E_R$$

We estimate the cost of performing a manual impact assessment by assuming that each node will have to be rechecked by the safety engineer to assess its content/state validity (we do not distinguish between the two w.r.t. their cost). Therefore, we compute the cost of manual impact assessment on a GSN safety case with E elements as follows:

$$Cost_{MIA} = K_R \times E$$

We are aware that there may be more intelligent ways to do a manual impact assessment on a safety case, but since this may vary from one person to another depending on their expertise, and since this hasn't been systematically studied, we have gone with a worst-case scenario approximation.

Next, we describe the LMS system before using it with *MMINT-A*.

10.2 The Lane Management System (LMS)

LMS describes a lane management system from the automotive domain. It is considered an ADAS (Advanced Driver Assistance System) system, which is safety critical and subject to the ISO 26262 standard.

The Software Requirements Specification (SRS) document for LMS from [Blazy *et al.*(2014)] describes LMS as consisting of several subcomponents. These subcomponents include a Lane Centering

System (LCS), a Lane Departure Warning System (LDWS) and a Lane Keeping System (LKS). The LMS is designed to be placed in automobiles as a safety feature with the goal of keeping the driver's vehicle in or near the centre of their lane to avoid crashes caused by drivers who become distracted and therefore inattentive to what lane they are in. The LDWS will issue warnings to the driver when the system determines that a lane change was unintentional. The LCS and LKS will work together to take control of the vehicle and adjust to a driver-defined centre of the lane. The overall system will make use of output data from several already-developed subsystems including: Camera Sensing Subsystem, Image Processing Subsystem, Vehicle State Estimation System, Path Prediction Subsystem, Driver Interface Subsystem and a Supervisory Control System. The LMS will be able to take control of the vehicle's braking and steering systems; however, the system will not be able to accelerate. Finally, the LMS will work at speeds above five miles per hour only.

LMS is comprised of 1 class diagram, 4 sequence diagrams and 4 state machines (a total of 9 system design models), which are all provided in Appendix B along with traceability tables that show the trace links between them. Figure B.3, specifically, shows the various subcomponents and how they are related. A megamodel of the LMS system in *MMINT* can be seen in Figure 10.1.

In the following section, we explain how a high level hazard analysis was done on LMS in order to identify safety goals used to create a safety case for the system.

10.3 LMS Safety Case

The first step in creating a safety case is identifying the system hazards by means of a hazard analysis activity. A *hazard*, as defined in ISO 26262 [ISO(2011)], is a potential source of harm (physical injury or damage to the health of persons) caused by malfunctioning behaviour (failure or unintended behaviour of an item with respect to its design intent) of the item (in this case the LMS system under consideration).

As described in ISO 26262 [ISO(2011)], a hazard analysis and risk assessment method is typically used to identify and categorize hazardous events of items and to specify safety goals and ASILs related to the prevention or mitigation of the associated hazards in order to avoid unreasonable risk. For this, the item is evaluated with regard to its functional safety. Safety goals and their assigned ASIL are determined by a systematic evaluation of hazardous events. The ASIL is determined by considering the estimate of the impact factors, i.e. *severity*, *probability of exposure* and *controllability*. It is based on the item's functional behaviour; therefore, the detailed design of the item does not necessarily need to be known.

We have conducted a high level hazard analysis for LMS and identified the following two hazards:

- System Hazard 1 (H1): Failing to notify driver when LMS fails (Vehicle Hazard: Unintended operation of vehicle feature)
- System Hazard 2 (H2): LMS prevents driver overriding control of steering (Vehicle Hazard: Vehicle feature prevents driver from controlling the vehicle)

Next, and in order to determine the appropriate ASIL level assigned to each of the hazards, we assign levels for each of their severity, probability of exposure, and controllability. The severity represents an estimate of the potential harm in a particular driving situation, while the probability of exposure is determined by the corresponding situation. The controllability rates how easy or difficult it is for the driver or other road traffic participant to avoid the considered accident type in the considered operational situation.

Based on guidance in the standard and expert opinion on the LMS system, we have assigned the two hazards the levels shown in Figure 10.2. We then used the ASIL determination table given by the standard and shown in Figure 10.3, to compute the associated ASIL levels.

Next, we construct a GSN representation of a safety case for LMS as seen in Figure 10.4[1].

We start by defining a top level goal "*G0: The LMS System Safety Goals are satisfied*". This is decomposed into:
- "*G1: The set of safety goals is complete*", which is a claim about the completeness of the system safety goals,
- "*G2: The LMS system notifies driver if it fails*", which is a safety goal associated with hazard **H1** (see the context node *C0*), and assigned an ASIL A, and
- "*G3: LMS always allows user to override and take control*", which is a safety goal associated with hazard **H2** (see the context node *C1*), and assigned an ASIL B.

G1 cannot be further decomposed, but is linked to the solution "*Sn0: HAZOP reviewed by expert*" via the strategy "*S1: HAZOP Analysis by technical expert*". The solution node *Sn0* will point to the document containing the HAZOP (Hazard and Operability Analysis) results which would have been used to derive the safety goals and can be used to assess their completeness by an expert review.
G2 is decomposed into:
- "*G4: if the LMS fails, prior to shutting off it will alert the driver*", and
- "*G5: LMS can detect failure in any of its subsystems*".

This is done via the strategy "*S2: Decomposition over procedure (check failure and then notify)* " which ensures coverage of both steps (checking failure and notification). Note that while *G4* inherits its parent goal's ASIL level (A), *G5* is assigned a higher ASIL (B) as it also supports goal *G7* which has an ASIL B.

Also of particular interest is the decomposition of "*G4: If the LMS fails, prior to shutting off, it will alert the driver (ASIL A)*". An ASIL decomposition strategy "(*S4: Decompose over user alerts*)" is used and follows the rules of decomposing an ASIL A into an ASIL A and QM (refer to Figure 6.6).

We continue the goal decomposition until we have a set of leaf goals that can no longer be

[1]The *MMINT-A* editor does not properly display the entire text in each of the GSN elements. To help with this problem, a version of the safety case was created in the Astah GSN tool and is presented in Figure 10.5. Note that Astah does not support the addition of strategies to connect leaf nodes with solutions, something we enforce by our AC metamodel, and therefore these strategies do not appear in the Astah version.

decomposed, but can be directly linked to supporting evidence via solution nodes. We do, however, enforce the use of strategies to connect leaf goals to supporting solutions, as we believe this gives a stronger argument for the linkage. Various types of supporting evidence are referred to in the solution nodes, ranging from expert opinion, to test results, to the supporting software requirements specification document, etc.

Upon completion of the goal decomposition and solution assignment, we end up with a 100-node assurance case for the LMS system[2].

Next, we create links from the LMS system design models to the safety case. The table in Figure 10.6 shows a traceability matrix linking the LMS safety case elements to the LMS class diagram model in Figure B.3. The trace links were created manually by linking safety case elements to parts of the class diagram that they were deemed to refer to (e.g., there is a link between the goal **G0** and the class *LMSSystem* and the goal **G12** and the operation *TurnOff()*) in the LDWS class. We assume that the safety case has links only to the LMS class diagram, which in turn links to the other LMS system design models that the LMS megamodel is comprised of. Such traceability is then handled by the assurance case impact assessment approach and taken into account when performing the megamodel slicing. We are aware that this is not a complete traceability, in the sense that there could be other links from the safety case to the LMS class diagram; we have only selected the apparent ones for the sake of demonstrating the approach.

In the following section, we demonstrate the use of the assurance case impact assessment approach on various change scenarios in the LMS system.

10.4 LMS Change Impact Assessment Scenarios

In this section, we consider three types of change scenarios on the LMS system and their effect on the corresponding safety case by using our impact assessment approach.

10.4.1 Change Scenario 1: Direct System Change

The first scenario we consider is a direct system change. In this scenario, a change is done on a system element that is *directly* related to the safety case. Specifically, we consider the system design change involving the removal of the audible alerts mechanism.

Change definition: Removal of audible alerts mechanism due to the deletion of the class "AudibleAlarm" in the LMS CD model in Figure B.3. As seen in the traceability table in Figure 10.6, the "AudibleAlarm" class is directly linked to the goals **G20** and **G18**, as well as the solutions **Sn12** and **Sn3** in the safety case.

Result of running MMINT-A: Figure 10.7 demonstrates the resulting annotated safety case after running the tool on this change. Goals **G20** and **G18**, as well as the solutions **Sn12** and **Sn3**

[2]Note that we consider ASILs as their own separate nodes.

are all marked for revision, as they directly refer to the AudibleAlarm. Other nodes are marked as recheck or reuse following the assurance case slicing rules presented in Section 9.3.

Figure 10.8 displays the annotation statistics for this scenario. A total of two goals (7% of the total number of goals) and two solution nodes (10% of the total number of solutions) are marked for revision. A total of three goals (11% of the total number of goals) are marked for recheck state, and five strategies (17% of the total number of strategies) and two ASIL nodes (8% of the total number of ASIL nodes) are marked for recheck content. This means a total of 14 elements are affected by this change, forming 14% of the assurance case. Finally, 21 goals (80% of the total number of goals), 23 strategies (82% of the total number of strategies), 18 solutions (90% of the total number of solutions), two contexts (100% of the total number of context nodes), and 22 ASIL nodes (91% of the total number of ASIL nodes) are marked for reuse. This gives the safety engineer an idea of the level of impact that this particular change can have on the safety case.

Figure 10.9 shows a portion of the backward traceability table produced in *MMINT-A*. Using this, the safety engineer is able to identify the source of a particular impact (annotation). For example, goal **G20** is marked as revise, and the element that triggered that annotation is the `lms_cd.Class AudibleAlarm`, which is the AudibleAlarm class in the LMS class diagram model. Goal **G5** is marked as recheck state, and the element that triggered that annotation is the `lms_sc.Basic Goal G18`, which is goal **G18** in the LMS safety case.

Cost analysis: Figure 10.10 shows the cost analysis for this scenario. There are a total number of four elements marked for revision, and 10 elements marked for recheck. As stated in Section 10.1, we assume the cost of revision $K_V = 2$ and the cost of recheck $K_R = 1$ (i.e., revisions cost double the amount of rechecks). Therefore, the automated CIA cost (Cost_CIA) is computed to equal 18, while the cost of performing a manual impact assessment (Cost_MIA) on a total of 100 elements is 100, an 82% cost reduction using our approach in this scenario.

10.4.2 Change Scenario 2: Indirect System Change

The second scenario we consider is an indirect system change. In this scenario, a change in a system element that is *indirectly* related to the safety case occurs (i.e., an element in a system model that does not have a direct trace link to the safety case, but can affect the safety case elements through indirect links from other elements in the system megamodel). Specifically, we consider the design change involving the modification of the path prediction system.

Change definition: Modification of path prediction system due to a change in the class "PathPredictionSystem" in the LMS CD model in Figure B.3. As seen in Figure 10.6, the "PathPredictionSystem" class is *not* directly linked to any element in the LMS safety case. However, due to traceability with other system models directly linked to the safety case, the megamodel slicing approach on the system will pick up the effect of this change on the safety case.

Result of running MMINT-A: Figure 10.11 demonstrates the resulting annotated safety case after running the tool given this change. As expected, we do not see any nodes marked for revision

as there is no direct impact on the safety case, however, we do see recheck annotations.

Figure 10.12 displays the annotation statistics for this scenario. No nodes are marked for revision. A total of 14 goals (53% of the total number of goals) and five solution nodes (25% of the total number of solutions) are marked for recheck content. A total of one goal (3% of the total number of goals) is marked for recheck state. This means a total of 20 elements are affected by this change, forming 20% of the assurance case. Finally, 11 goals (42% of the total number of goals), 28 strategies (100% of the total number of strategies), 15 solutions (75% of the total number of solutions), two contexts (100% of the total number of context nodes), and 24 ASIL nodes (100% of the total number of ASIL nodes) are marked for reuse. This gives the safety engineer an idea of the level of impact this change can have on the safety case.

Figure 10.13 shows a portion of the backward traceability report that the safety engineer can use to examine the source of impact for each annotation as needed.

Cost analysis: Figure 10.14 shows the cost analysis for this scenario. There are a total number of zero elements marked for revision and 20 elements marked for recheck. Therefore, the automated CIA cost (Cost_CIA) is computed to equal 20, while the cost of performing a manual impact assessment (Cost_MIA) on a total of 100 elements in 100, an 80% cost reduction using our approach in this scenario.

10.4.3 Change Scenario 3: Design Space Exploration

The third scenario that we consider is the activity of performing a *design space exploration*. In this scenario, the safety engineer uses our approach and tool to explore the effects of different design changes on the safety case before communicating back to the system design engineers to make any modifications to the system. Specifically, we consider the potential effect of modifying the *TurnOff* functions in the LCS and LKS subsystems on the overall system safety case.

Change definition: Modified LCS TurnOff versus modified LKS TurnOff. As seen in the LMS to safety case traceability table in Figure 10.6, the LKS TurnOff function is linked to both **G11** and **Sn5**, whereas the LCS TurnOff function is linked to both **G13** and **Sn10**.

Result of running MMINT-A: Figure 10.15 and Figure 10.16 demonstrate the resulting annotated safety cases after running the tool given the LCS and LKS TurnOff changes, respectively. As seen in Figure 10.17, modifying the LCS TurnOff function causes 16 goals and seven solutions to be marked for recheck content, while, as seen in Figure 10.18, modifying the LKS TurnOff function causes 18 goals and nine solutions to be marked for recheck content. They both cause one goal only to be marked for recheck state. A change in LCS TurnOff affects a total of 24 elements or 24% of the assurance case, while a change in LKS TurnOff affects 28 elements or 28% of the assurance case. On the other hand, modifying the LCS TurnOff function allows for the reuse of nine goals, 28 strategies, 13 solutions, two context nodes and 24 ASIL nodes, while modifying the LKS TurnOff function allows for the reuse of seven goals, 28 strategies, 11 solutions, two context nodes and 24 ASIL nodes.

Figure 10.19 and Figure 10.20 demonstrate the resulting backward traceability tables which show the source of impact for each annotation in the LCS and LKS TurnOff change scenarios, respectively.

Cost analysis: Figures 10.21 and 10.22 show the cost analysis for LCS and LKS TurnOff changes, respectively. For the LCS TurnOff case, there are a total number of zero elements marked for revision and 24 elements marked for recheck. Therefore, the automated CIA cost (Cost_CIA) is computed to equal 24, while the cost of performing a manual impact assessment (Cost_MIA) on a total of 100 elements in 100, a 76% cost reduction using our approach in this case. On the other hand, for the LKS TurnOff case, there are a total number of zero elements marked for revision and 28 elements marked for recheck. Therefore, the automated CIA cost (Cost_CIA) is computed to equal 28, while the cost of performing a manual impact assessment (Cost_MIA) on a total of 100 elements in 100, a 72% cost reduction using our approach in this case. The safety engineer may consider this, along with the impact of each change described above, when exploring system design changes.

10.5 Chapter Summary

The goal of this chapter was to conduct a case study on an automotive subsystem in order to determine whether our assurance case impact assessment approach improves efficiency. We associate efficiency with the reduction of the cost of performing an assurance case impact assessment given system design changes. We have considered the Lane Management System (LMS) from the automotive domain, which is comprised of a total of nine system design models. We performed a high level hazard analysis to identify hazards, associate them with safety goals, and assign them ASIL levels. We used the result of the hazard analysis to construct a safety case for the system which is comprised of 100 elements.

To assess our approach, we considered three different types of scenarios: direct system change, indirect system change, and design space exploration.

In the direct system change scenario, a deletion of an element in the system directly linked to the safety case caused a total of 14 elements (14% of the assurance case) to be affected by this change. The automated CIA cost is 18, compared to a manual impact assessment cost of 100, an 82% cost reduction using our approach.

In the indirect system change scenario, a modification to an element in the system indirectly linked to the safety case caused a total of 20 elements (20% of the assurance case) to be affected by this change. Interestingly, this is greater than the number of elements affected by a direct system change, which demonstrates that some indirect system changes could have a larger impact on the assurance case than direct system changes. The automated CIA cost is 20, compared to a manual impact assessment cost of 100, an 80% cost reduction using our approach.

Finally, in the design space exploration scenario, we looked at how a change in the same function in different subsystems can have different effects on the safety case. This type of analysis could be used by the safety engineer when communicating back to the system design engineers to make design

decisions. For example, a change in the TurnOff method in one subsystem caused a total of 24 (24% of the assurance case) to be affected, while a change in the same method for a different subsystem affected a total of 28 elements (28% of the assurance case). The automated CIA cost is 24 and 28 in each of these cases, compared to a manual impact assessment cost of 100, a 72-76% cost reduction using our approach.

Observations. In all three scenarios, we have observed that the number of elements that the safety engineer needs to inspect using our approach is significantly reduced compared to a manual approach. Although we over-approximate the cost of a manual impact assessment approach, there is still a clear reduction in the cost associated with the effort required to perform the safety case change impact assessment, an indicator that this approach may have significant impact in practice. In the future, we plan to better understand how a manual impact assessment cost can be computed, and to consider additional usage scenarios.

We are aware that the level of traceability (among system models and between system models and the safety case) affects the results of the CIA, namely, traceability precision and CIA precision are correlated. However, since traceability is often expensive (creating, discovering and/or maintaining trace links is difficult in practice), there is a tradeoff to be considered.

Another aspect we have observed while conducting this case study is that this approach can be useful in the case of more than one assurance case for the same system. Because of the way different assurance cases are structured, some may be more robust against change than others. In the future, it would be interesting to define a metric for assessing robustness of an assurance case, and use our approach to identify it.

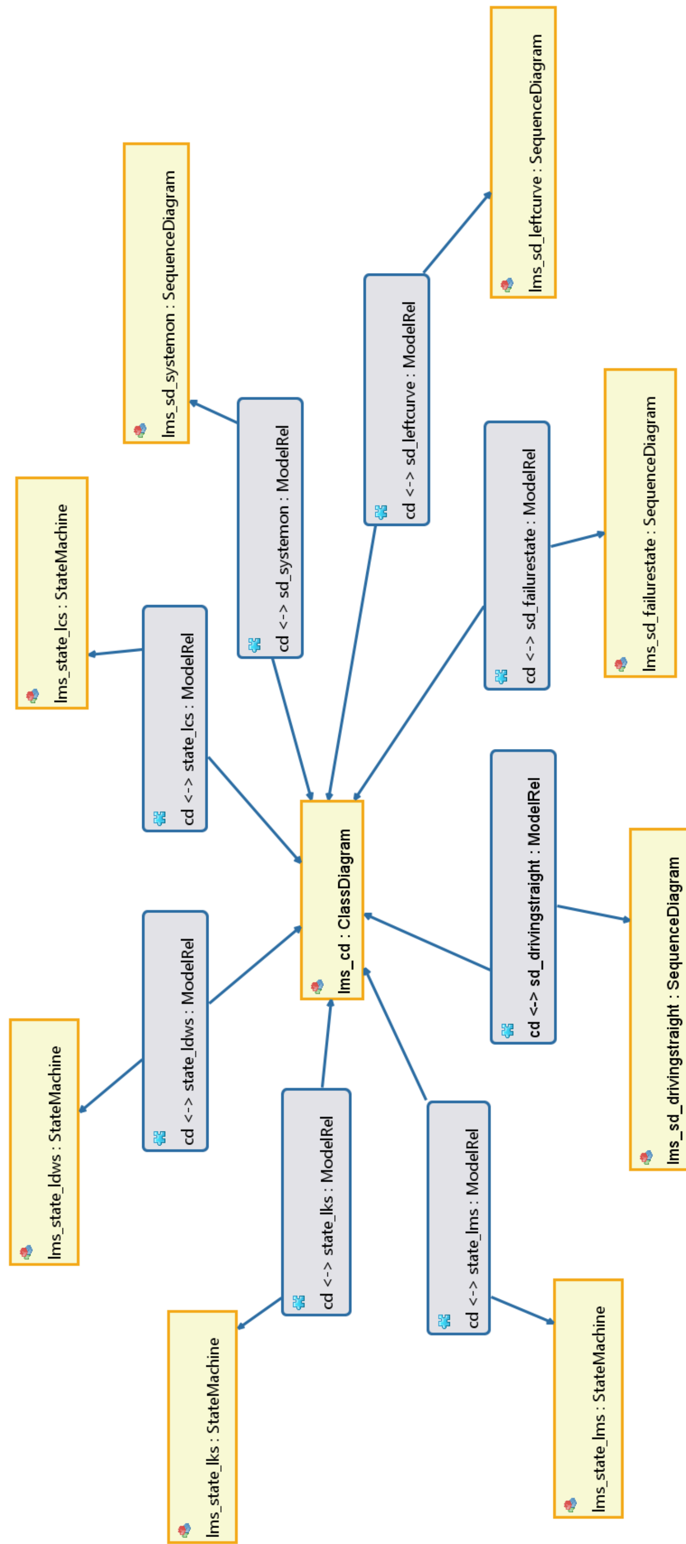


Figure 10.1: LMS system megamodel in *MMINT*.

id	Vehicle Hazard	System Hazard	Sev	Exp	Con	ASIL
H1	Unintended operation of vehicle feature	Failing to notify driver when LMS fails	S1	E4	C2	A
H2	Vehicle feature prevents driver from controlling the vehicle	LMS prevents driver overriding control of steering	S3	E2	C3	B

Figure 10.2: LMS Hazard Analysis

Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Figure 10.3: ASIL determination table from [ISO(2011)]

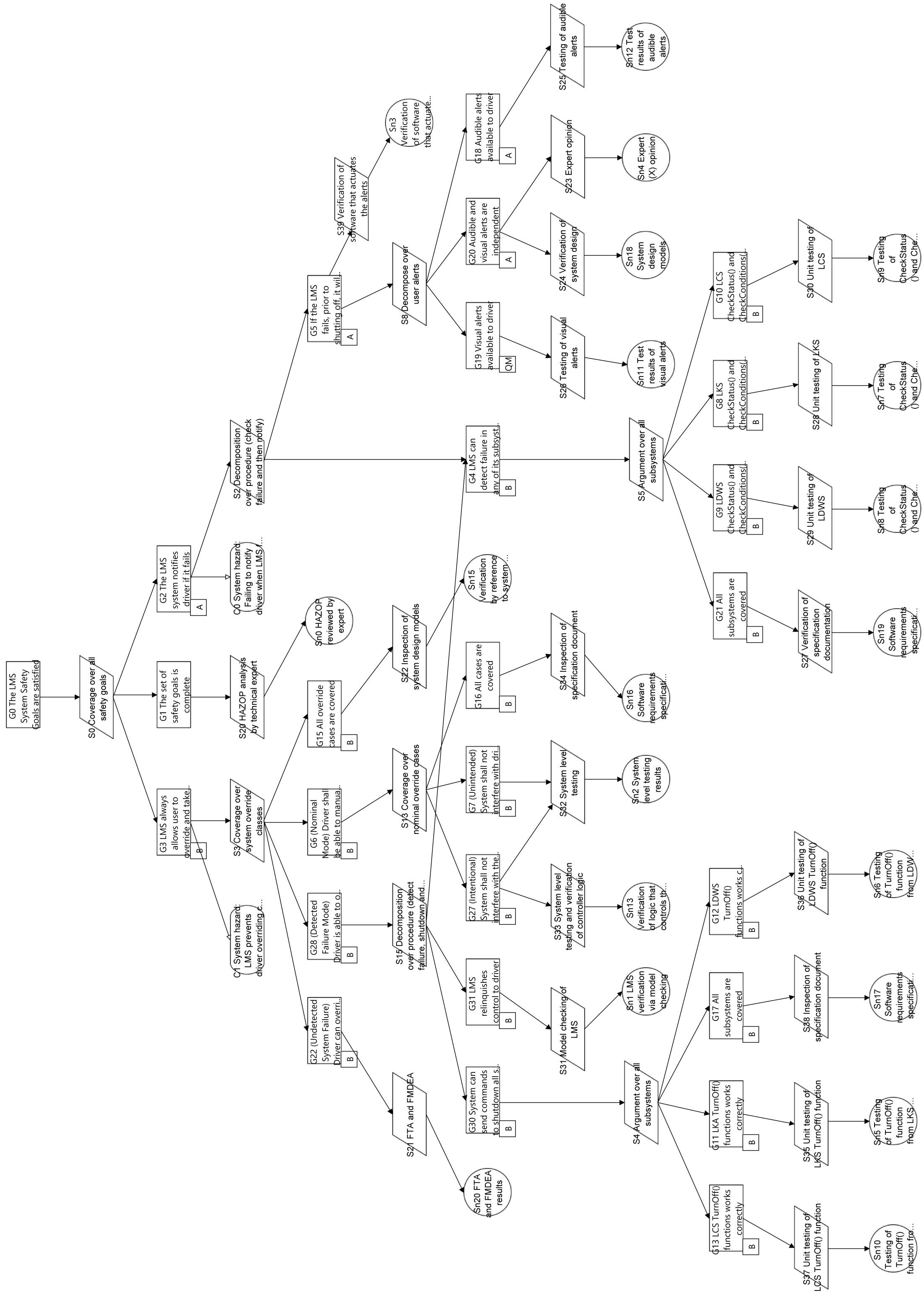


Figure 10.4: LMS Safety Case in *MMINT-A*

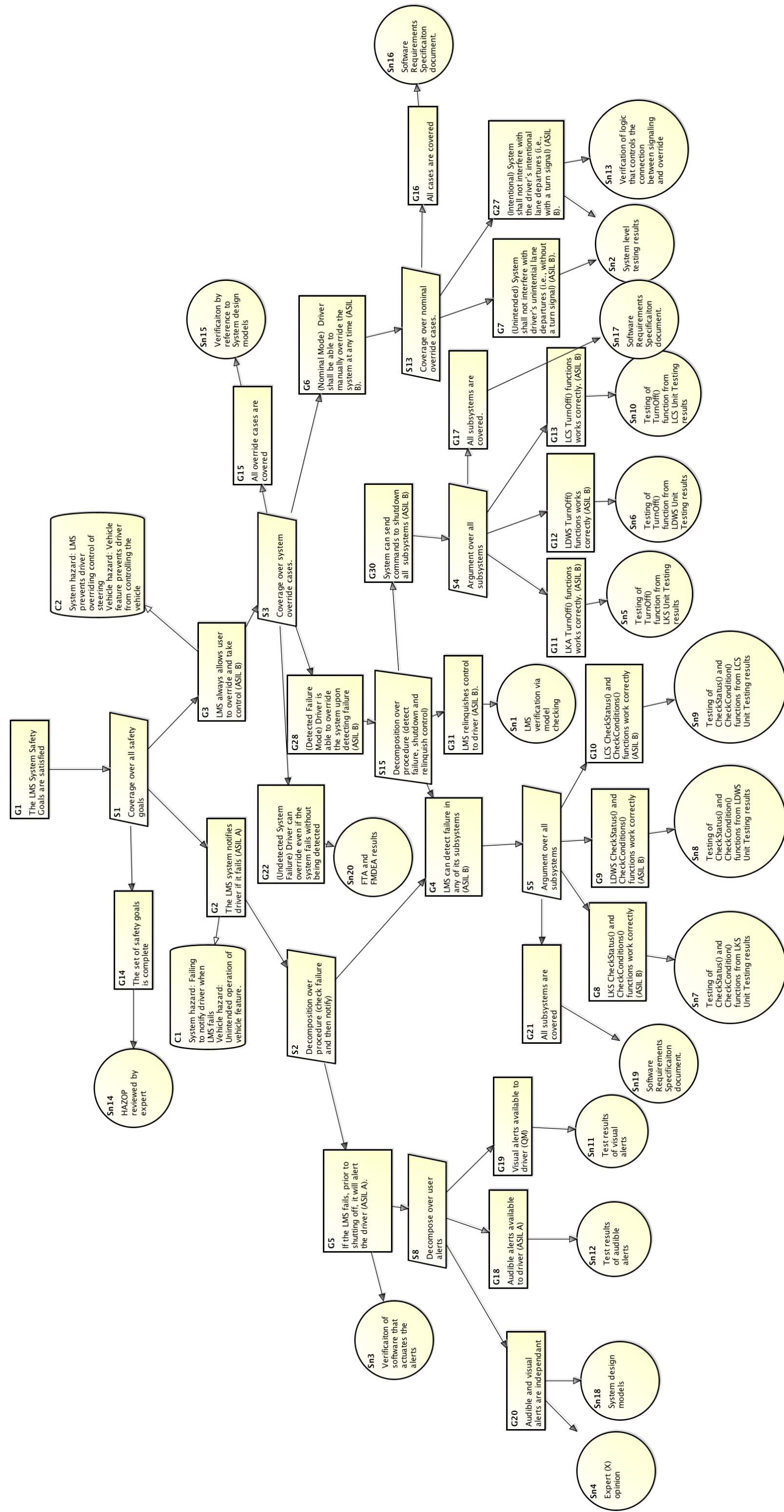


Figure 10.5: LMS Safety Case in Astah GSN

			From: Class Diagram Element											
			Class LMSSystem	Class AudibleAlarm	Class VisualAlarm	Operation CheckStatus() (Class LKS)	Operation CheckConditions() (Class LKS)	Operation CheckStatus() (Class LDWS)	Operation CheckConditions() (Class LDWS)	Operation CheckStatus() (Class LCS)	Operation CheckConditions() (Class LCS)	Operation TurnOff() (Class LKS)	Operation TurnOff() (Class LDWS)	Operation TurnOff() (Class LCS)
To: Safety Case Element	Goal	G0	√											
	Goal	G2	√											
	Goal	G3	√											
	Goal	G5	√											
	Goal	G4	√											
	Goal	G22	√											
	Goal	G28	√											
	Goal	G6	√											
	Goal	G20		√	√									
	Goal	G18		√										
	Goal	G19			√									
	Goal	G8				√	√							
	Goal	G9						√	√					
	Goal	G10								√	√			
	Goal	G31	√											
	Goal	G11										√		
	Goal	G12											√	
	Goal	G13												√
	Solution	Sn12		√										
	Solution	Sn11			√									
	Solution	Sn7				√	√							
	Solution	Sn8						√	√					
	Solution	Sn9								√	√			
	Solution	Sn5										√		
	Solution	Sn6											√	
	Solution	Sn10												√
	Solution	Sn3		√	√									

Figure 10.6: LMS Safety Case to LMS CD Traceability Matrix

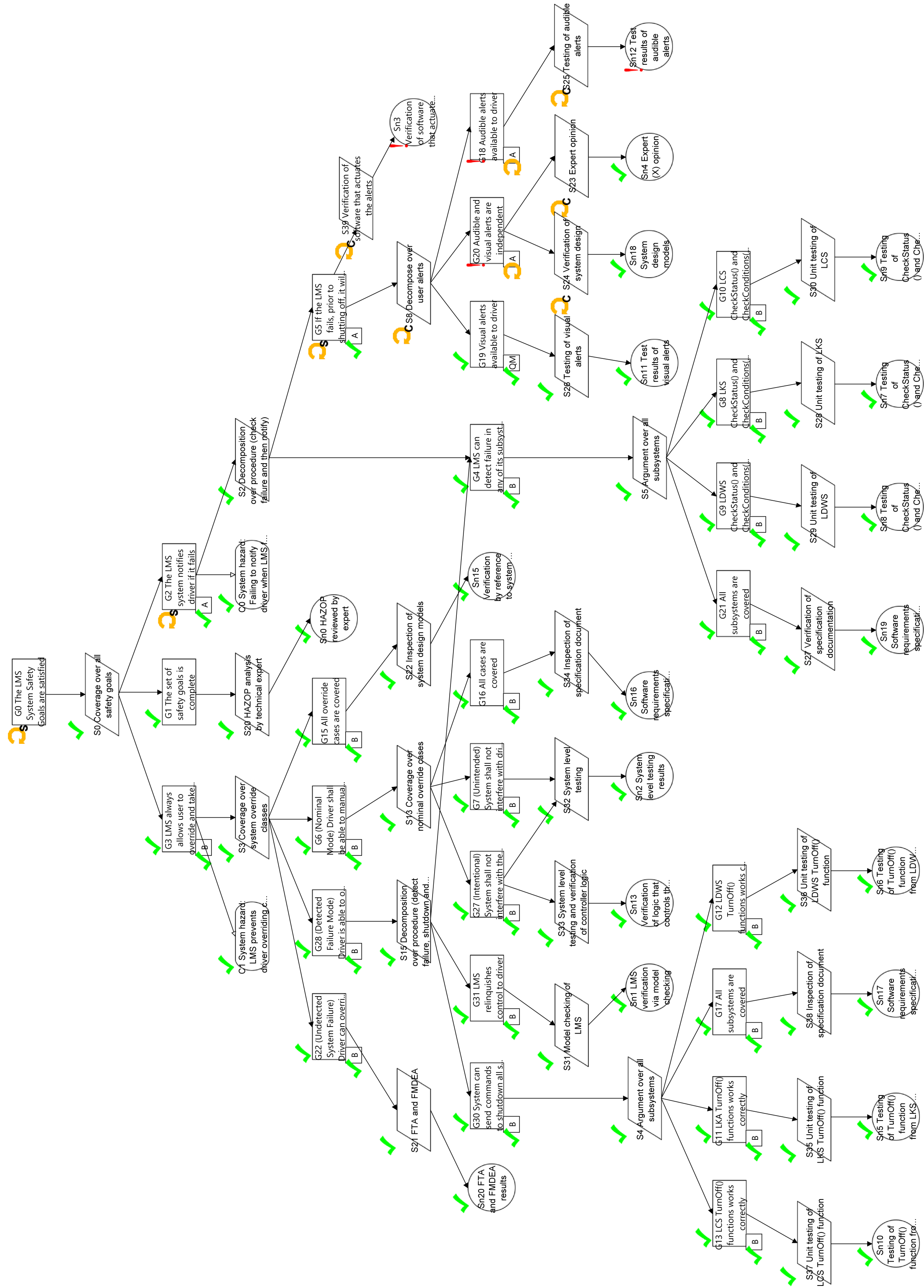


Figure 10.7: LMS Annotated Safety Case after Change Scenario 1

	Goals	Strategies	Solutions	Context	Justifications	Assumptions	ASILs
Revise	2 (7%)	0 (0%)	2 (10%)	0 (0%)			0 (0%)
Recheck Content	0 (0%)	5 (17%)	0 (0%)	0 (0%)			2 (8%)
Recheck State	3 (11%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Reuse	21 (80%)	23 (82%)	18 (90%)	2 (100%)			22 (91%)
Total	26	28	20	2	0	0	24

Figure 10.8: Statistics report for Scenario 1

	Impact Type	Impact Source
Goal G0	RecheckState	lms_sc.Basic Goal G18
Goal G1	Reuse	Not applicable.
Goal G2	RecheckState	lms_sc.Basic Goal G18
Goal G3	Reuse	Not applicable.
Goal G5	RecheckState	lms_sc.Basic Goal G18
Goal G4	Reuse	Not applicable.
Goal G22	Reuse	Not applicable.
Goal G28	Reuse	Not applicable.
Goal G6	Reuse	Not applicable.
Goal G15	Reuse	Not applicable.
Goal G20	Revise	lms_cd.Class AudibleAlarm
Goal G18	Revise	lms_cd.Class AudibleAlarm
Goal G19	Reuse	Not applicable.
Goal G21	Reuse	Not applicable.
Goal G8	Reuse	Not applicable.
Goal G9	Reuse	Not applicable.
Goal G10	Reuse	Not applicable.
Goal G31	Reuse	Not applicable.
Goal G30	Reuse	Not applicable.
Goal G7	Reuse	Not applicable.
Goal G27	Reuse	Not applicable.
Goal G16	Reuse	Not applicable.
Goal G11	Reuse	Not applicable.
Goal G12	Reuse	Not applicable.

Figure 10.9: Backward traceability report for Scenario 1

	Revised No. (E_V)	Recheck No. (E_R)	K_V	K_R	Cost_CIA	Total No. (N)	Cost_MIA
Value	4	10	2	1	18	100	100

Figure 10.10: Cost analysis for Scenario 1

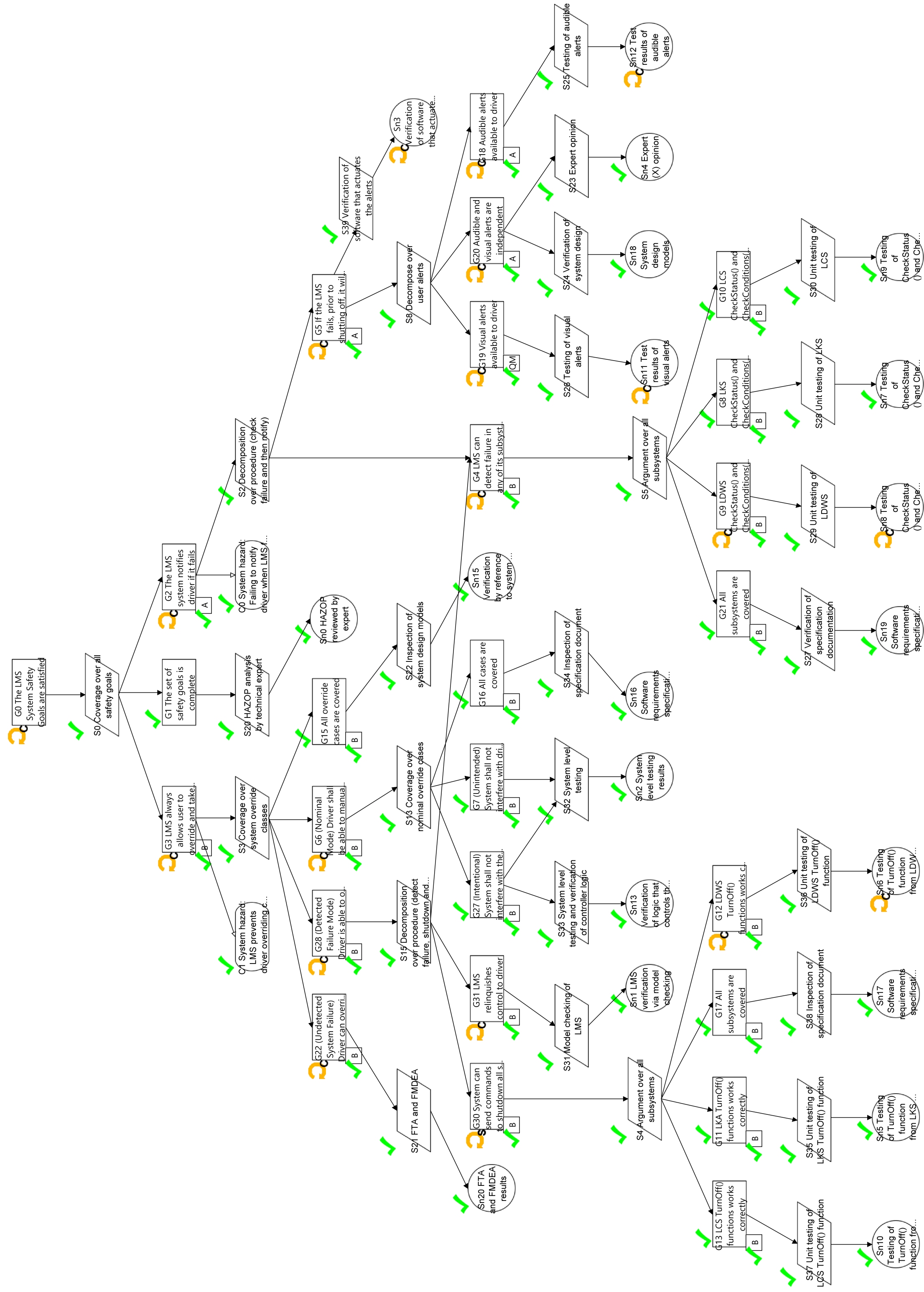


Figure 10.11: LMS Annotated Safety Case after Change Scenario 2

	Goals	Strategies	Solutions	Context	Justifications	Assumptions	ASILs
Revise	0 (0%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Recheck Content	14 (53%)	0 (0%)	5 (25%)	0 (0%)			0 (0%)
Recheck State	1 (3%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Reuse	11 (42%)	28 (100%)	15 (75%)	2 (100%)			24 (100%)
Total	26	28	20	2	0	0	24

Figure 10.12: Statistics report for Scenario 2

	Impact Type	Impact Source
Goal G0	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G1	Reuse	Not applicable.
Goal G2	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G3	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G5	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G4	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G22	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G28	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G6	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G15	Reuse	Not applicable.
Goal G20	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm, lms_
Goal G18	RecheckContent	lms_cd.Class Alarm
Goal G19	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm
Goal G21	Reuse	Not applicable.
Goal G8	Reuse	Not applicable.
Goal G9	RecheckContent	lms_cd.Class LaneDepartureWarningSystem, lms_cd.Class LaneD
Goal G10	Reuse	Not applicable.
Goal G31	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G30	RecheckState	lms_sc.Basic Goal G12
Goal G7	Reuse	Not applicable.
Goal G27	Reuse	Not applicable.
Goal G16	Reuse	Not applicable.
Goal G11	Reuse	Not applicable.
Goal G12	RecheckContent	lms_cd.Class LaneDepartureWarningSystem, lms_cd.Class LaneD

Figure 10.13: Backward traceability report for Scenario 2

	Revised No. (E_V)	Recheck No. (E_R)	K_V	K_R	Cost_CIA	Total No. (N)	Cost_MIA
Value	0	20	2	1	20	100	100

Figure 10.14: Cost analysis for Scenario 2

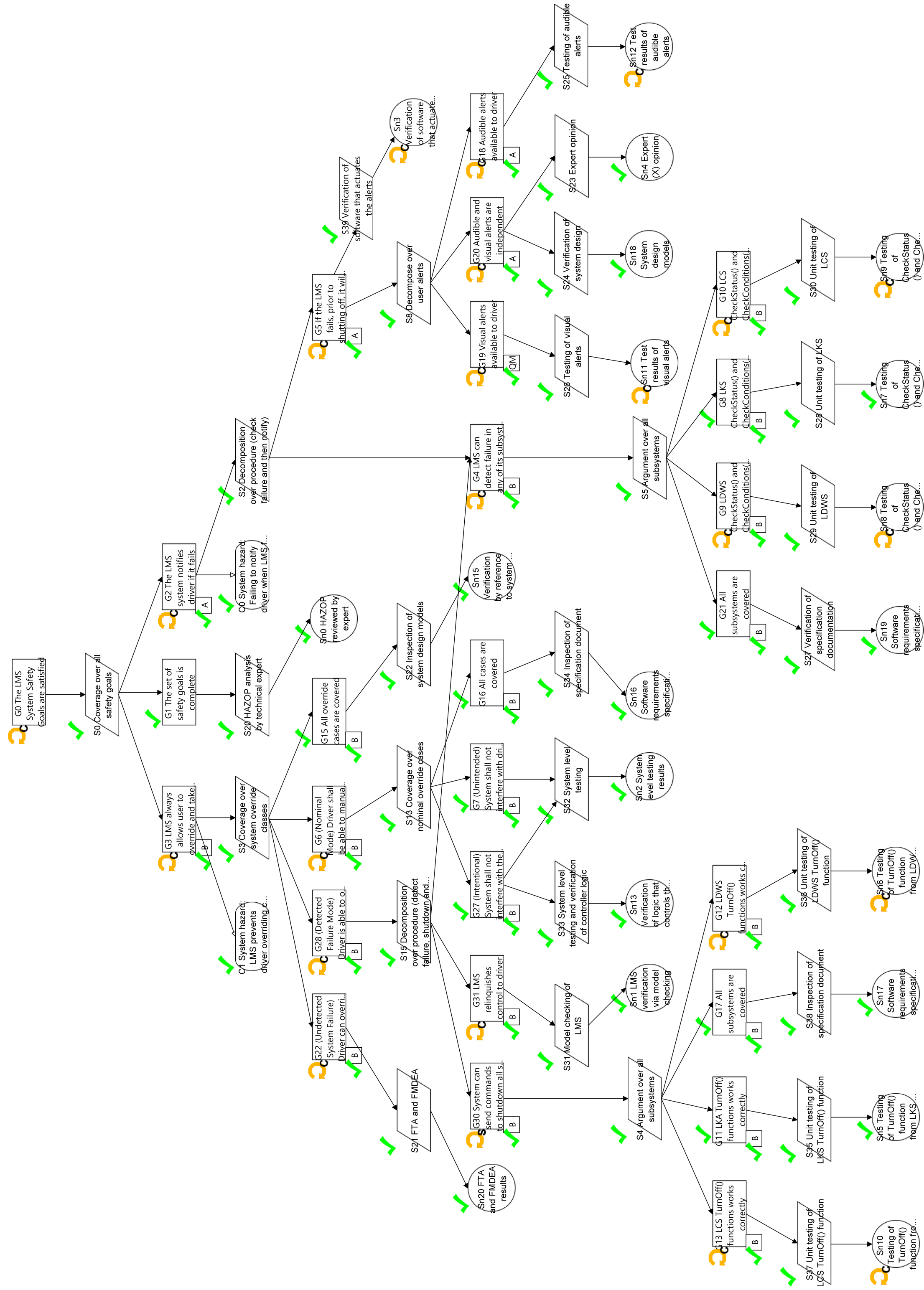


Figure 10.15: LMS Annotated Safety Case after Change Scenario 3 - LCS TurnOff()

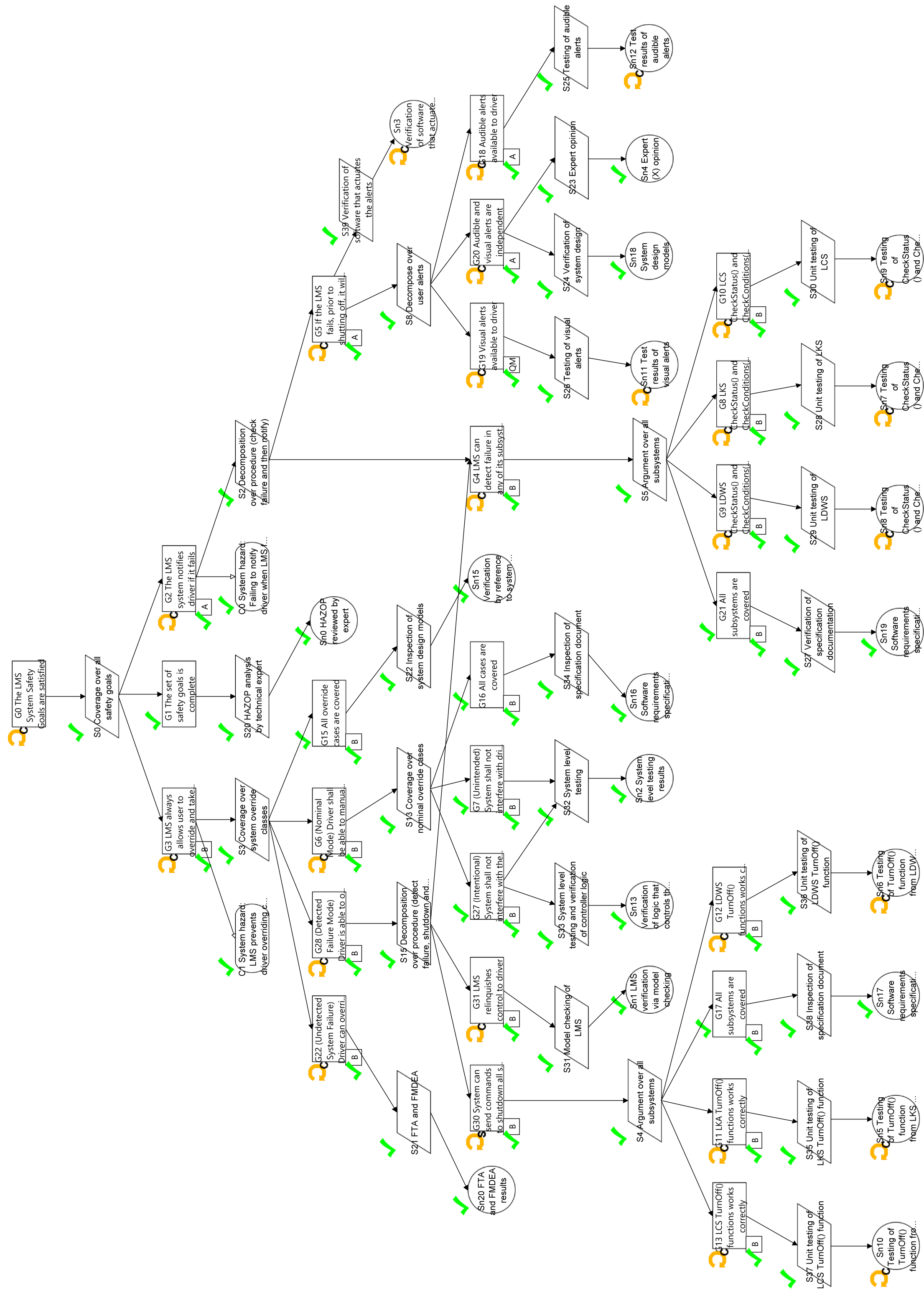


Figure 10.16: LMS Annotated Safety Case after Change Scenario 3 - LKS TurnOff()

	Goals	Strategies	Solutions	Context	Justifications	Assumptions	ASILs
Revise	0 (0%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Recheck Content	16 (61%)	0 (0%)	7 (35%)	0 (0%)			0 (0%)
Recheck State	1 (3%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Reuse	9 (34%)	28 (100%)	13 (65%)	2 (100%)			24 (100%)
Total	26	28	20	2	0	0	24

Figure 10.17: Statistics report for Scenario 3 - LCS TurnOff()

	Goals	Strategies	Solutions	Context	Justifications	Assumptions	ASILs
Revise	0 (0%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Recheck Content	18 (69%)	0 (0%)	9 (45%)	0 (0%)			0 (0%)
Recheck State	1 (3%)	0 (0%)	0 (0%)	0 (0%)			0 (0%)
Reuse	7 (26%)	28 (100%)	11 (55%)	2 (100%)			24 (100%)
Total	26	28	20	2	0	0	24

Figure 10.18: Statistics report for Scenario 3 - LKS TurnOff()

	Impact Type	Impact Source
Goal G0	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G1	Reuse	Not applicable.
Goal G2	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G3	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G5	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G4	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G22	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G28	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G6	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G15	Reuse	Not applicable.
Goal G20	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm, lms
Goal G18	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm
Goal G19	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm
Goal G21	Reuse	Not applicable.
Goal G8	Reuse	Not applicable.
Goal G9	RecheckContent	lms_cd.Class LaneDepartureWarningSystem, lms_cd.Class LaneD
Goal G10	RecheckContent	lms_cd.Class LaneCenteringSystem, lms_cd.Class LaneCentering
Goal G31	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G30	RecheckState	lms_sc.Basic Goal G12
Goal G7	Reuse	Not applicable.
Goal G27	Reuse	Not applicable.
Goal G16	Reuse	Not applicable.
Goal G11	Reuse	Not applicable.
Goal G12	RecheckContent	lms_cd.Class LaneDepartureWarningSystem, lms_cd.Class LaneD

Figure 10.19: Traceability report for Scenario 3 - LCS TurnOff()

	Impact Type	Impact Source
Goal G0	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G1	Reuse	Not applicable.
Goal G2	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G3	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G5	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G4	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G22	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G28	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G6	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G15	Reuse	Not applicable.
Goal G20	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm, lms
Goal G18	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm
Goal G19	RecheckContent	lms_cd.Class Alarm, lms_cd.Class Alarm, lms_cd.Class Alarm
Goal G21	Reuse	Not applicable.
Goal G8	RecheckContent	lms_cd.Class LaneKeepingSystem, lms_cd.Class LaneKeepingSy
Goal G9	RecheckContent	lms_cd.Class LaneDepartureWarningSystem, lms_cd.Class LaneD
Goal G10	RecheckContent	lms_cd.Class LaneCenteringSystem, lms_cd.Class LaneCentering
Goal G31	RecheckContent	lms_cd.Composition, lms_cd.Composition, lms_cd.Composition
Goal G30	RecheckState	lms_sc.Basic Goal G11
Goal G7	Reuse	Not applicable.
Goal G27	Reuse	Not applicable.
Goal G16	Reuse	Not applicable.
Goal G11	RecheckContent	lms_cd.Class LaneKeepingSystem, lms_cd.Class LaneKeepingSy
Goal G12	RecheckContent	lms_cd.Class LaneDepartureWarningSystem, lms_cd.Class LaneD

Figure 10.20: Traceability report for Scenario 3 - LKS TurnOff()

	Revised No. (E_V)	Recheck No. (E_R)	K_V	K_R	Cost_CIA	Total No. (N)	Cost_MIA
Value	0	24	2	1	24	100	100

Figure 10.21: Cost analysis for Scenario 3 - LCS TurnOff()

	Revised No. (E_V)	Recheck No. (E_R)	K_V	K_R	Cost_CIA	Total No. (N)	Cost_MIA
Value	0	28	2	1	28	100	100

Figure 10.22: Cost analysis for Scenario 3 - LKS TurnOff()

Part V

Conclusions & Future Work

Chapter 11

Conclusion

This chapter concludes the thesis. Section 11.1 recaps the contributions of the thesis, and Section 11.2 provides an overview of future research directions that will both extend this research work and address limitations of this thesis.

11.1 Summary of Contributions

In many domains, including the automotive domain, where assurance cases are used as a means to argue about critical qualities (e.g., safety) of systems, including the software that runs on them, it is a manual and costly activity to maintain assurance cases as systems evolve. In this thesis, we have presented an approach, based on model management, to address the problem of managing assurance cases in model based software systems. The approach aids the safety engineer in creating an evolved assurance case for the evolved system, by providing guidance as to which parts of the original assurance case can be reused, and which ones need to be rechecked or revised. This is done by constructing a workflow of model management operators to achieve this task. The technique presented in this thesis can be further used to construct other workflows to address other assurance case management scenarios. The rest of this section discusses the technical contributions in this thesis.

In Part I of this thesis (Chapter 1), we first introduce our ideas for using model management as a way to capture various compliance related scenarios. Specifically, we set out a research agenda that includes using model management as a level of abstraction to structure and manage the complexity of the compliance problem, using model relationships to express traceability between artifacts used for compliance checking, and using model management operators to (semi-)automate compliance management activities such as assurance case evolution due to system design changes.

Part II of this thesis is about megamodel management; the foundation for the assurance case

management work in Part III. After presenting a running example and relevant background material on modeling and model management in Chapter 2, we presented our tool, *MMINT*, which allows interactive model management in Chapter 3. We then presented preliminaries in Chapter 4, which are required to understand how we formalize collections of models and relationships between them as megamodels, and our basic megamodel management operators and techniques to address heterogeneity, which are used in the rest of the thesis. In Chapter 5, we presented an approach for slicing heterogenous megamodels, which is used as a basis for the assurance case impact assessment approach described in Chapter 7.

Part III of this thesis is about assurance case management. After presenting relevant background material on standards and assurance cases in Chapter 6, we presented a model management approach for assurance case reuse due to system evolution in Chapter 7. First, we defined a generic model management framework for addressing this problem. Then, we identified and specified the model management operators needed for a semi-automated solution for the problem and presented an algorithm for it. Next, we evaluated the generic framework and proposed solution by instantiating it for ISO 26262 vehicle safety cases with the KAOS goal modeling language used for expressing assurance cases. Finally, we applied this instantiation to the power sliding door automotive system for validation.

In Chapter 8, we built on our work in the previous chapter, which applies to assurance cases in general and ensures soundness, i.e., it does not miss any elements that are impacted. Yet, because the approach is conservative, it can flag elements as impacted when they are not, resulting in "false positives". We proposed to use knowledge about the system models, the safety case language and the standard under consideration, to improve the precision of our approach, thus reducing unnecessary effort by the safety engineer. Specifically, the main contributions of this chapter are a model-based approach for impact assessment on GSN safety cases used with ISO 26262, as well as six techniques for improving the precision of the impact assessment approach.

Part IV of this thesis is about tool support and validation. In Chapter 9, we presented our tool *MMINT-A* that can, in the context of model-driven development, assess the impact of system changes on their assurance cases. To achieve this, *MMINT-A* implements an impact assessment algorithm from Chapters 7 and 8 and incorporates a graphical assurance case editor, an annotation mechanism, and two summary tables for the assessment results. For validation, we demonstrated the usage and advantages of our approach and tool, *MMINT-A*, on a Lane Management System case study from the automotive domain in Chapter 10. The case study suggests that the approach may have significant impact in practice. We intend to validate this claim further on a larger scale system and with the input of an assurance engineer from industry.

11.2 Future Work

In Chapter 1 we set out a research agenda that includes the following:
1. Using model management as a level of abstraction to structure and manage the complexity of the compliance problem.
2. Using model relationships to express traceability between artifacts used for compliance checking.
3. Using model management operators to (semi-)automate compliance management activities such as standard and artifact evolution or extracting relevant portions of standards.

Specifically, we identified a set of compliance-related research problems that model management can be used to solve. We summarize them in Table 11.1 for reference.

Problem	Description
P1	Creating a general model of compliance that defines explicit relationships between the compliance artifacts.
P2	Reusing evidence and other assurance artifacts due to standard or product evolution.
P3	Extracting relevant parts of a standard or a system for checking compliance.
P4	Effort reduction of compliance to multiple standards.
P5	Lifting compliance from a single product to a product line.
P6	Identifying relationships between standards.

Table 11.1: Summary of compliance management problems from Chapter 1.

In this thesis, we have proposed solutions to each of **P1** and **P2**. We have also proposed heterogenous megamodeling operators, such as **map**, **filter** and **reduce**, which can be used in a workflow to address **P3**. **P4** and **P6** can also be addressed with our proposed model management operators and tool, if standards are described in a model-based fashion, and explicit traceability between them is defined. **P5**, however, requires a careful treatment of variability in order to make use of our proposed approaches for product lines. We leave these problems as future work, some of which we have already started addressing.

In the rest of this section, we discuss some limitations of our work, how it can be further improved and how it can feed into future directions of research.

11.2.1 Limitations and Improvements

Relaxing Assumptions. The assurance case change impact analysis approach relies on a set of assumptions, that we aim to relax. For example, we assume that a complete and correct traceability relationship is always given between the system models and the assurance case. This is not always the case in practice, and we plan to study approaches that enable automated trace link discovery [Guo *et al.*(2017)] to help address this. We also assume that we are always given a valid assurance case (GSN-like artifact) that is correct (e.g., w.r.t. goal decomposition), and complete (e.g., well-formedness rules hold such as each leaf goal being grounded in evidence). We plan to incorporate

approaches that can deal with incomplete assurance cases, and that can assess validity of the input assurance case. Finally, we assume that the change in the system is known to us (via modified, deleted and added elements). If this is not explicitly known, we plan to use approaches such as model differencing techniques to help explicate this delta.

A Human-in-the-loop Approach. The assurance case change impact assessment approach we offer is meant as a starting point for assessing the impact of a change on the assurance case. The annotated assurance case that the techniques and tool produce is meant to be given to the safety engineer for further processing. The safety engineer should consider the elements marked for revision and revise them, making new changes either to the system models or to the assurance case, and possibly triggering another change impact assessment. We therefore see this as an incremental human-in-the-loop approach. Furthermore, we envision a set of guidelines that can be used to support the safety engineer in making the revisions. These could, for example, consider prioritizing revision order (e.g., based on ASIL level, semantics of the assurance case (e.g., context is inherited by lower nodes), or by considering imposing an order on the assurance case). Other heuristics could also be derived based on experience in working with assurance cases by the engineers themselves.

Design Space Exploration and Project Planning. We have demonstrated an approach for safety case impact assessment due to system changes, primarily with the aim of reducing effort (increasing reuse) in performing CIA. However, we believe that our approach can be used for impact assessment in general, and not just for safety case co-evolution. One application for this is in design space exploration, to enable answering what-if questions about the impact of changes on assurance cases, with the ultimate goal of understanding how to develop software so that frequent changes do not significantly affect assurance cases (or such changes are contained). For example, one might want to understand the effect of replacing a component with another on the safety case. Here we can use the reporting provided by our tool on percentages of elements marked for revision vs. those reusable, to understand the extent of impact of a particular change verses another on the safety case. The question may not always be just "what if?", but also "why'?'. For example, given that an element is affected by a change, why was it marked for revision. Here, understanding which slicing rules were fired, and over which parts of the model, to affect this element, are of interest. We have an implementation of this analysis, but it might be useful to express it purely as a rule-based question.

We have also been thinking about using the approach for project planning. In this case, knowing which teams are responsible for implementing parts of the systems (via some organizational structure models for example), we can use this information (with the backward traceability feature in our tool) to identify which teams need to be contacted to address changes causing impact on the assurance case.

Addressing Additions. Currently, our impact assessment approach addresses the effect of adding components in the system on the existing parts of the safety case. However, it currently cannot

address how adding components can potentially require additions to the safety case. In this context, two types of additions, namely, known and unknown additions, should be considered. Known additions can come from a 150% representation of the system, known design decisions that can be selected from, known variations (turning features on/off) in product lines, and known variation/design spaces. Unknown additions are harder to tackle, as their traceability to the original system is often unknown and techniques for trace link discovery [Guo *et al.*(2017)] will need to be used. We also need to distinguish between ad-hoc additions vs. composition. The former relates to incrementally adding new parts to the system, and therefore needing to incrementally asses the safety case, while the latter deals with adding a well-defined module, which could have its own safety case, and needing to compose safety cases. We plan to study these kinds of additions further, their connection to emergent behaviour, and propose approaches for addressing them in the future.

Exploiting System Design Patterns. We would like to understand whether our approach can detect certain changes in the system design which change not only the system functionality but its level of integrity. For example, consider the "redundancy pattern", where a component such as the redundant switch in our PSD example is added. We would like to study if it is possible to identify this case as a redundancy change by witnessing two paths to the actuator (one via the VS ECU and one via the redundant switch), and how this can be exploited for impact assessment.

Change Assurance Cases. Our impact assessment approach can guide the creation of a Change Assurance Case: an argument for the changes made to the original safety case, providing evidence for such an argument. For example, our approach can support a revise marking in a safety case by linking the element to the appropriate counterparts in the system megamodel that caused this marking to be computed. We would like to explore what such a Change Assurance Case looks like. Specifically, for particular changes (e.g., feature addition or integrity addition, etc.) we might be able to give a change assurance case skeleton. In the example of increasing system integrity, we could provide a change assurance case that needs to show at least two sub-claims: 1. A claim that no functionality is affected (i.e., the behaviour of the system before and after the change is equivalent), which could be done via regression testing to show that w.r.t. the same test suite, we obtain the same set of behaviour. 2. A claim that the likelihood of the identified hazards occurring has decreased (due to adding integrity) and that other hazards have not been introduced. We see our impact analysis approach as being one piece of evidence for this change argument.

Confidence Modeling. We would like to augment our approach to handle a confidence model on top of safety cases. That is, we would like to assess the impact of changes not just on the safety case elements themselves but on the confidence level we assign them (e.g., based on expert opinion) and on the safety case as a whole.

Generalization. There are a few areas we can address to generalize the approach further. First, broadening from UML models to other design models, such as Simulink, as well as to non-design

models and artifacts (e.g., code, test cases, hazard-analysis, FMEA, FTA). Second, considering not only the product level claims, but the process level claims. This can be done by making use of a process model as a source of knowledge, which may also include an "organizational structural model" which could link to an "escalation path" to identify the right persons/teams to contact for certain changes. Third, the approach can be generalized from the product-level to be used at the product-line level. Here, and specifically in delta product lines, the impact assessment approach could be used iteratively, taking the appropriate deltas into account, to understand the impact of changes on the product-line level. Finally, an interesting direction would be in understanding how more general changes (e.g., refactoring) that can be thought of as transformations with intent on the system models, and how certain transformations affect the safety cases in certain ways, with the goal of identifying change patterns related to these transformations.

An Improved Cost Metric. We use a cost metric to asses how much we managed to reduce the effort of change impact assessment using our approach. The cost analysis can be smarter than our current cost function, by considering lower level nodes vs. higher level nodes (e.g., it is more costly to revise something at a lower level in the assurance case argument as it has more impact on things above it in the tree). We can also associate cost assessment with the different evidence types, where depending on the type of evidence, reuse may be more valuable.

Tool support, Validation and Verification We are still actively working on extending and improving our tool *MMINT-A*. We plan to incorporate the improvement techniques discussed in this thesis and validate their effectiveness. We also aim to add support for OMG's Structured Assurance Case Metamodel (SACM [OMG(2015)]) and conduct a usability study with our industrial partner to identify specific barriers in adopting *MMINT-A*. These may include integration with other tools (e.g., for compliance checking) as well as support for assuring product lines.

We intend to evaluate our framework on a real world case study, ideally using input from an actual Assurance Engineer. We expect such a study to be done with our industrial partner. We also plan to study the reuse of assurance case components under scenarios other than system evolution. For example, the evolution of standards such as ISO 26262 will impact the assurance case constructed to show compliance of the system to the original standard. Also, safety goal change due to the use of a different hazard analysis technique could occur and will also impact the assurance case. Another scenario is the reuse of assurance case components between similar systems or within a product line of systems. We would also like to study a case where emergent behaviour occurs due to adding new elements to the system [Johnson(2006), Fiadeiro(1996)].

Although the algorithms presented in this thesis have been analysed with respect to various properties, we aim to add confidence in our approaches by using theorem provers to check properties such as soundness, termination, minimality, etc. This is particularly important to support their usage in a safety critical context.

11.2.2 Future Research Directions

Advanced Model-Based Techniques to support Rigour, Automation and Reuse in Industry. Industry has started paying attention to MDE as an approach that can bring many advantages to traditional software development, but there is much room for improvement. Some longer term goals are to discover new areas where MDE can offer advantages, and to help in "bridging the gap" between research and practice to realize such advantages. As demonstrated in this thesis, model-based workflows can be constructed to address issues such as impact assessment, evolution and reuse. This work has developed a workflow for one scenario, but many others can be considered as described in Chapter 1. The goal would be to enrich MDE with techniques from other areas, increasing its applicability and impact in the real world. One direction is the use of AI with MDE. Specifically, using Natural Language Processing (NLP) to discover and improve traceability among the many artifacts in a software ecosystem, including requirements, design models, tests results, etc. Some work has been done in this area [Guo *et al.*(2017)], but it has not been connected with MDE and has not been yet used in critical software systems which require an additional artifact, namely an assurance case. This direction of research will lead to rigorous automated approaches resulting in huge cost savings at companies where currently much of the trace link discovery and maintenance effort is done in an ad-hoc and manual fashion. Another goal would be to study how to scale some of the model-based approaches developed in this thesis from individual products to product lines, which typically reflect the state of practice in industry.

The research done in the requirements engineering community, specifically, on goal modelling, and studying the relationship with assurance cases is a direction worth considering. In this context, the work on requirements evolution over time [Grubb and Chechik(2016)], and how some of these techniques can be applied to assurance cases as (safety) goals change over time would be interesting to pursue. A final direction in this area is considering uncertainty at design time [Famelis and Chechik(2017)], and how this can be handled in critical systems with respect to reasoning about their assurance in an incremental manner until the uncertainty is eventually resolved.

Software Compliance and Certification for Complex Software Intensive Systems. In practice, software and system manufacturers lack a consistent and effective set of guidelines as to what constitutes acceptable evidence of software quality, and how to achieve it. Typically, critical software-intensive systems are certified on the basis of the *process* used to develop them. While a a good process is indeed necessary for producing dependable software, it is not sufficient. Software certification should also be based on evidence obtained from the *product*. A goal of this work would be to study what kind of evidence is sufficient for software certification, and how different kinds of evidence may be combined into a correct and complete assurance argument. While this work has begun to address this from a system safety perspective, the intention is to generalize findings beyond safety to areas such as privacy and security. This work has also been focused on the automotive domain and the ISO 26262 standard for functional safety of road vehicles. It is worth generalizing to other domains such as aerospace and medical devices. Here, the objectives would be to study the

software development lifecycles and the regulations and standards that govern software development in these domains, and to develop techniques to check compliance and aid in *software certification*, *incremental certification* as features and components are added, and *recertification* as systems and regulations change and evolve. The interplay between software qualities, such as between safety and security, is an interesting yet understudied area, which would be interesting to explore. The goal would be to develop sound ways to model this interplay and reason about it as systems evolve. Software certification is further complicated by the use of AI techniques in critical software systems. Also, with the advent of Smart Systems, the Internet of Things (IoT), and Agile methodologies, it is crucial that software certification advances to cover these issues and help ensure high quality of software systems.

Bibliography

[A. Kitchenham(2007)] A. Kitchenham, B. (2007). Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE-2007-01, EBSE.

[Abdullah *et al.*(2010)] Abdullah, N. S., Sadiq, S., and Indulska, M. (2010). Emerging Challenges in Information Systems Research for Regulatory Compliance Management. In *International Conference on Advanced Information Systems Engineering*, pages 251–265. Springer.

[Adelard(2018)] Adelard (2018). Claims, Arguments and Evidence (CAE). https://www.adelard.com/asce/choosing-asce/cae.html.

[Aiello *et al.*(2014)] Aiello, M. A., Hocking, A. B., Knight, J. C., and Rowanhill, J. C. (2014). SCT: A Safety Case Toolkit. In *IEEE International Symposium on Software Reliability Engineering Workshops*, pages 216–219.

[Aizenbud-Reshef *et al.*(2006)] Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., and Shaham-Gafni, Y. (2006). Model traceability. *IBM Systems Journal*, **45**(3), 515–526.

[Allan *et al.*(1998)] Allan, J., Williams, J., Gander-Miller, G., Turner, M., Ballantyne, T., and Harvey, J. (1998). Safety Case Production. *WIT Transactions on The Built Environment*, **37**.

[Althammer *et al.*(2009)] Althammer, E., Schoitsch, E., Eriksson, H., and Vinter, J. (2009). The DECOS Concept of Generic Safety Cases - A Step towards Modular Certification. In *35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 537–545.

[Ankrum and Kromholz(2005)] Ankrum, T. S. and Kromholz, A. H. (2005). Structured Assurance Cases: Three Common Standards (Presentation). In *Ninth IEEE International Symposium on High-Assurance Systems Engineering*, pages 99–108.

[Arendt *et al.*(2010)] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135.

[AssureNote(2018)] AssureNote (2018). AssureNote. https://github.com/AssureNote/AssureNote.

[Attwood *et al.*(2011)] Attwood, K., Chinneck, P., Clarke, M., and et. al (2011). GSN Community Standard Version 1. Technical report, Origin Consulting (York) Limited.

[Bandur and McDermid(2015)] Bandur, V. and McDermid, J. (2015). Informing Assurance Case Review Through a Formal Interpretation of GSN Core Logic. In *International Conference on Computer Safety, Reliability, and Security*, pages 3–14.

[Barry(2011)] Barry, M. R. (2011). CertWare: A Workbench for Safety Case Production and Analysis. In *IEEE Aerospace Conference*, pages 1–10.

[Bernstein(2003)] Bernstein, P. A. (2003). Applying Model Management to Classical Meta Data Problems. In *Conference on Innovative Data Systems Research*, volume 2003, pages 209–220.

[Beydeda *et al.*(2005)] Beydeda, S., Book, M., Gruhn, V., *et al.* (2005). *Model-driven software development*, volume 15. Springer.

[Bézivin *et al.*(2004)] Bézivin, J., Jouault, F., and Valduriez, P. (2004). On the Need for Megamodels. In *Proc. of OOPSLA/GPCE Workshops*.

[Bézivin *et al.*(2005a)] Bézivin, J., Jouault, F., and Touzet, D. (2005a). An Introduction to the Atlas Model Management Architecture. Technical Report 05.01.

[Bézivin *et al.*(2005b)] Bézivin, J., Jouault, F., Rosenthal, P., and Valduriez, P. (2005b). Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. Springer.

[Bjornander *et al.*(2012)] Bjornander, S., Land, R., Graydon, P., Lundqvist, K., and Conmy, P. (2012). A Method to Formally Evaluate Safety Case Arguments Against a System Architecture Model. In *International Symposium on Software Reliability Engineering Workshops*, pages 337 – 342.

[Blazy *et al.*(2014)] Blazy, B., DeLine, A., Frey, B., and Miller, M. (2014). Software Requirements Specification (SRS): Lane Management System.

[Bloomfield and Bishop(2010)] Bloomfield, R. and Bishop, P. (2010). Safety and Assurance Cases: Past, Present and Possible Future – an Adelard Perspective. In *Safety-Critical Systems: Problems, Process and Practice*, pages 51–67. Springer.

[Blouin *et al.*(2011)] Blouin, A., Combemale, B., Baudry, B., and Beaudoux, O. (2011). Modeling Model Slicers. In *International Conference on Model Driven Engineering Languages and Systems*, pages 62–76. Springer.

[Blouin *et al.*(2015)] Blouin, A., Combemale, B., Baudry, B., and Beaudoux, O. (2015). Kompren: Modeling and Generating Model Slicers. *Journal of Software and Systems Modeling*, **14**(1), 321–337.

[Borg *et al.*(2016)] Borg, M., de la Vara, J. L., and Wnuk, K. (2016). Practitioners' Perspectives on Change Impact Analysis for Safety-Critical Software – A Preliminary Analysis. In *International Conference on Computer Safety, Reliability, and Security*, pages 346–358. Springer.

[Brambilla *et al.*(2012)] Brambilla, M., Cabot, J., and Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool.

[Brunel and Cazin(2012)] Brunel, J. and Cazin, J. (2012). Formal Verification of a Safety Argumentation and Application to a Complex UAV System. In *Formal Verification of a Safety Argumentation and Application to a Complex UAV System*, pages 307–318. Springer.

[Brunet *et al.*(2006)] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzadeh, M. (2006). A Manifesto for Model Merging. In *International Workshop on Global Integrated Model Management*, pages 5–12. ACM.

[Calder *et al.*(2003)] Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature Interaction: a Critical Review and Considered Forecast. *Computer Networks*, **41**(1), 115–141.

[Calinescu *et al.*(2017)] Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M. U., Habli, I., and Kelly, T. (2017). Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. *IEEE Transactions on Software Engineering*, pages 1–30.

[Cârlan *et al.*(2017)] Cârlan, C., Barner, S., Diewald, A., Tsalidis, A., and Voss, S. (2017). ExplicitCase: Integrated Model-based Development of System and Safety Cases. In *International Conference on Computer Safety, Reliability, and Security*, volume 10489 LNCS, pages 52 – 63.

[Cassano *et al.*(2016)] Cassano, V., Singh, N., Grigorova, S., Patcas, L., Kokaly, S., Lawford, M., Maibaum, T., and Wassyng, A. (2016). Making Sense of ISO 26262: Some Structured Views. Submitted to Reliability Engineering and System Safety journal.

[Cheng *et al.*(2018)] Cheng, J., Goodrum, M., Metoyer, R., and Cleland-Huang, J. (2018). How do practitioners perceive assurance cases in safety-critical software systems? *arXiv preprint arXiv:1803.08097*.

[Chowdhury *et al.*(2017)] Chowdhury, T., Lin, C.-W., Kim, B., Lawford, M., Shiraishi, S., and Wassyng, A. (2017). Principles for systematic development of an assurance case template from iso 26262. In *Software Reliability Engineering Workshops, 2017 IEEE International Symposium on*, pages 69–72. IEEE.

[Cimatti *et al.*(2015)] Cimatti, A., De Long, R., Marcantonio, D., and Tonetta, S. (2015). Combining MILS with Contract-Based Design for Safety and Security Requirements. In *International Conference on Computer Safety, Reliability, and Security*, volume 9338 of *LNCS*, pages 264 – 276.

[Clark(2011)] Clark, T. (2011). A General Model-Based Slicing Framework. In *Proceedings of Workshop on Composition and Evolution of Model Transformations*.

[Conrad *et al.*(2012)] Conrad, M. *et al.* (2012). Artifact-centric compliance demonstration for iso 26262 projects using model-based design. In *GI-Jahrestagung*, pages 807–816. Citeseer.

[Cruanes *et al.*(2013)] Cruanes, S., Hamon, G., Owre, S., and Shankar, N. (2013). Tool Integration with the Evidential Tool Bus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, volume 7737, pages 275–294.

[Dardenne *et al.*(1993)] Dardenne, A., Van Lamsweerde, A., and Fickas, S. (1993). Goal-directed requirements acquisition. *Science of Computer Programming*, **20**(1), 3–50.

[de la Vara(2014)] de la Vara, J. L. (2014). Current and Necessary Insights into SACM: An Analysis Based on Past Publications. In *IEEE International Workshop on Requirements Engineering and Law (RELAW)*, pages 10–13. IEEE.

[de la Vara and Panesar-Walawege(2013)] de la Vara, J. L. and Panesar-Walawege, R. K. (2013). Safetymet: A Metamodel for Safety Standards. In *International Conference on Model Driven Engineering Languages and Systems*, pages 69–86. Springer.

[de la Vara *et al.*(2016a)] de la Vara, J. L., Borg, M., Wnuk, K., and Moonen, L. (2016a). An Industrial Survey of Safety Evidence Change Impact Analysis Practice. *IEEE Transactions on Software Engineering*, **42**(12), 1095–1117.

[de la Vara *et al.*(2016b)] de la Vara, J. L., Ruiz, A., Attwood, K., Espinoza, H., Panesar-Walawege, R. K., López, Á., del Río, I., and Kelly, T. (2016b). Model-based specification of safety compliance needs for critical systems: A holistic generic metamodel. *Information and Software Technology*, **72**, 16–30.

[Dean and Ghemawat(2008)] Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, **51**(1), 107–113.

[Denney and Pai(2017)] Denney, E. and Pai, G. (2017). Tool Support for Assurance Case Development. *Automated Software Engineering*, pages 1 – 65.

[Di Sandro *et al.*(2015)] Di Sandro, A., Salay, R., Famelis, M., Kokaly, S., and Chechik, M. (2015). MMINT: A Graphical Tool for Interactive Model Management. In *International Conference on Model Driven Engineering Languages and Systems(demo track)*.

[Diskin *et al.*(2010)] Diskin, Z., Xiong, Y., and Czarnecki, K. (2010). From state-to delta-based bidirectional model transformations. In *Theory and Practice of Model Transformations*, pages 61–76. Springer.

[Diskin *et al.*(2013)] Diskin, Z., Kokaly, S., and Maibaum, T. (2013). Mapping-Aware Megamodeling: Design Patterns and Laws. In *International Conference on Software Language Engineering*, pages 322–343.

[Diskin *et al.*(2014)] Diskin, Z., Wider, A., Gholizadeh, H., and Czarnecki, K. (2014). Towards a Rational Taxonomy for Increasingly Symmetric Model Synchronization. In *International Conference on Theory and Practice of Model Transformations*, pages 57–73. Springer.

[Eclipse(2018a)] Eclipse (2018a). Eclipse Modeling Framework(EMF). https://www.eclipse.org/modeling/emf/.

[Eclipse(2018b)] Eclipse (2018b). Sirius. https://www.eclipse.org/sirius/.

[Emmet and Cleland(2002)] Emmet, L. and Cleland, G. (2002). Graphical Notations, Narratives and Persuasion: A Pliant Systems Approach to Hypertext Tool Design. In *Proceedings of the thirteenth ACM conference on Hypertext and hypermedia*, pages 55–64. ACM.

[Erwig(1997)] Erwig, M. (1997). Functional Programming with Graphs. *ACM SIGPLAN Notices*, **32**(8), 52–65.

[Fahrenberg *et al.*(2014)] Fahrenberg, U., Acher, M., Legay, A., and Wąsowski, A. (2014). Sound Merging and Differencing for Class Diagrams. In *International Conference on Fundamental Approaches to Software Engineering*, pages 63–78. Springer.

[Falessi *et al.*(2011)] Falessi, D., Nejati, S., Sabetzadeh, M., Briand, L., and Messina, A. (2011). SafeSlice: A Model Slicing and Design Safety Inspection Tool for SysML. In *European Conference on Foundations of Software Engineering*, pages 460–463. ACM.

[Famelis and Chechik(2017)] Famelis, M. and Chechik, M. (2017). Managing design-time uncertainty. *Journal of Software and Systems Modeling*, pages 1–36.

[Favre *et al.*(2012)] Favre, J.-M., Lämmel, R., and Varanovich, A. (2012). *Modeling the Linguistic Architecture of Software Products*. Springer.

[Fenn *et al.*(2007)] Fenn, J. L., Hawkins, R. D., Williams, P., Kelly, T. P., Banner, M. G., and Oakshott, Y. (2007). The Who, Where, How, Why and When of Modular and Incremental Certification. In *2nd IET International Conference on System Safety*, pages 135–140. IET.

[Fiadeiro(1996)] Fiadeiro, J. L. (1996). On the Emergence of Properties in Component-Based Systems. In *International Conference on Algebraic Methodology and Software Technology*, pages 421–443. Springer.

[Fujita *et al.*(2012)] Fujita, H., Matsuno, Y., Hanawa, T., Sato, M., Kato, S., and Ishikawa, Y. (2012). DS-Bench Toolset: Tools for Dependability Benchmarking with Simulation and Assurance. In *International Conference on Dependable Systems and Networks*, pages 1–8.

[Fung *et al.*(2018)] Fung, N. L., Kokaly, S., Di Sandro, A., Salay, R., and Chechik, M. (2018). Mmint-a: A tool for automated change impact assessment on assurance cases. In *International Conference on Computer Safety, Reliability, and Security*, pages 60–70. Springer.

[Gacek *et al.*(2014)] Gacek, A., Backes, J., Cofer, D., Slind, K., and Whalen, M. (2014). Resolute: An Assurance Case Language for Architecture Models. In *ACM SIGAda Ada Letters*, pages 19–28.

[Gallina(2014)] Gallina, B. (2014). A Model-Driven Safety Certification Method for Process Compliance. In *International Symposium on Software Reliability Engineering*, pages 204–209. IEEE.

[Ghanavati *et al.*(2011)] Ghanavati, S., Amyot, D., and Peyton, L. (2011). A Systematic Review of Goal-Oriented Requirements Management Frameworks for Business Process Compliance. In *IEEE International Workshop on Requirements Engineering and Law (RELAW)*, pages 25–34. IEEE.

[Ghanavati *et al.*(2014)] Ghanavati, S., Rifaut, A., Dubois, E., and Amyot, D. (2014). Goal-Oriented Compliance with Multiple Regulations. In *Requirements Engineering Conference*, pages 73–82. IEEE.

[Gorski *et al.*(2012)] Gorski, J., Jarzebowicz, A., Miler, J., Witkowicz, M., Czyznikiewicz, J., and Jar, P. (2012). Supporting Assurance by Evidence-Based Argument Services. In *International Conference on Computer Safety, Reliability, and Security*, volume 7613 of *LNCS*, pages 417–426.

[Gotel and Finkelstein(1995)] Gotel, O. and Finkelstein, A. (1995). Contribution structures. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 100–107. IEEE.

[Groza and Marc(2014)] Groza, A. and Marc, N. (2014). Consistency Checking of Safety Arguments in the Goal Structuring Notation Standard. In *Intelligent Computer Communication and Processing*, pages 59–66.

[Grubb and Chechik(2016)] Grubb, A. M. and Chechik, M. (2016). Looking into the crystal ball: Requirements evolution over time. In *Requirements Engineering Conference (RE), 2016 IEEE 24th International*, pages 86–95. IEEE.

[GSN(2011)] GSN (2011). Goal Structuring Notation Working Group, "GSN Community Standard Version 1". http://www.goalstructuringnotation.info/.

[Guo *et al.*(2017)] Guo, J., Cheng, J., and Cleland-Huang, J. (2017). Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, pages 3–14. IEEE Press.

[Habli and Kelly(2008)] Habli, I. and Kelly, T. (2008). A Model-Driven Approach to Assuring Process Reliability. In *International Symposium on Software Reliability Engineering*, pages 7–16. IEEE.

[Habli *et al.*(2010)] Habli, I., Ibarra, I., Rivett, R. S., and Kelly, T. (2010). Model-Based Assurance for Justifying Automotive Functional Safety. Technical report, SAE.

[Hamou-Lhadj and Hamou-Lhadj(2007)] Hamou-Lhadj, A. and Hamou-Lhadj, A. (2007). Towards a Compliance Support Framework for Global Software Companies. In *Software Engineering Conference*, pages 182–192.

[Hawkins *et al.*(2015)] Hawkins, R., Habli, I., Kolovos, D., Paige, R., and Kelly, T. (2015). Weaving an Assurance Case from Design: A Model-Based Approach. In *High-Assurance Systems Engineering*, pages 110–117. IEEE.

[Heidenreich *et al.*(2011)] Heidenreich, F., Kopcsek, J., and Aßmann, U. (2011). Safe Composition of Transformations. *Journal of Object Technology*, **7**(10).

[Huhn and Zechner(2009)] Huhn, M. and Zechner, A. (2009). Analysing Dependability Case Arguments Using Quality Models. In *International Conference on Computer Safety, Reliability, and Security*, volume 5775 of *LNCS*, pages 118 – 131.

[ISO(2011)] ISO (2011). *ISO 26262: Road Vehicles – Functional Safety*. International Organization for Standardization. 1st version.

[Jaradat and Bate(2016)] Jaradat, O. and Bate, I. (2016). Systematic Maintenance of Safety Cases to Reduce Risk. In *International Conference on Computer Safety, Reliability, and Security*, pages 17–29. Springer.

[Johnson(2006)] Johnson, C. W. (2006). What Are Emergent Properties and How Do They Affect the Engineering of Complex Systems? *J. Reliability Engineering & System Safety*, **91**(12), 1475–1481.

[Kagdi *et al.*(2005)] Kagdi, H., Maletic, J. I., and Sutton, A. (2005). Context-Free Slicing of UML Class Models. In *International Conference on Software Maintenance*, pages 635–638. IEEE.

[Kawakami *et al.*(2016)] Kawakami, H., Ott, D., Wong, H. C., Dahab, R., and Gallo, R. (2016). ACBuilder: A Tool for Hardware Architecture Security Evaluation. In *Hardware Oriented Security and Trust*, pages 97–102.

[Kelly(1997)] Kelly, T. (1997). A Six-Step Method for the Development of Goal Structures. *York Software Engineering, Flixborough, UK.*

[Kelly and McDermid(2001a)] Kelly, T. and McDermid, J. (2001a). A Systematic Approach to Safety Case Maintenance. *Reliability Engineering & System Safety*, **1**(3), 271 – 284.

[Kelly and Weaver(2004)] Kelly, T. and Weaver, R. (2004). The Goal Structuring Notation – A Safety Argument Notation. In *Proceedings of the dependable systems and networks workshop on assurance cases.*

[Kelly(1998)] Kelly, T. P. (1998). *Arguing Safety: A Systematic Approach to Managing Safety Cases.* Ph.D. thesis, Univ. of York, UK.

[Kelly and McDermid(1997)] Kelly, T. P. and McDermid, J. A. (1997). Safety Case Construction and Reuse Using Patterns. In *International Conference on Computer Safety, Reliability, and Security*, pages 55–69. Springer.

[Kelly and McDermid(2001b)] Kelly, T. P. and McDermid, J. A. (2001b). A Systematic Approach to Safety Case Maintenance. *Reliability Engineering & System Safety*, **71**(3), 271–284.

[Khalil and Dingel(2013)] Khalil, A. and Dingel, J. (2013). Supporting the Evolution of UML Models in Model Driven Software Development: a Survey. Technical Report 602, School of Computing, Queen's University, Ontario, Canada.

[Kling *et al.*(2012)] Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., and Cabot, J. (2012). MoScript: A DSL for Querying and Manipulating Model Repositories. In *International Conference on Software Language Engineering*, pages 180–200. Springer.

[Kokaly(2017)] Kokaly, S. (2017). Managing assurance cases in model based software systems. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 453–456.

[Kokaly *et al.*(2016a)] Kokaly, S., Salay, R., Cassano, V., Maibaum, T., and Chechik, M. (2016a). A Model Management Approach for Assurance Case Reuse due to System Evolution. In *International Conference on Model Driven Engineering Languages and Systems*, pages 196–206.

[Kokaly *et al.*(2016b)] Kokaly, S., Salay, R., Sabetzadeh, M., Chechik, M., and Maibaum, T. (2016b). Model Management for Regulatory Compliance: a Position Paper. In *Modeling in Software Engineering workshop at International Conference on Software Engineering.*

[Kokaly *et al.*(2017)] Kokaly, S., Salay, R., Chechik, M., Lawford, M., and Maibaum, T. (2017). Safety Case Impact Assessment in Automotive Software Systems: An Improved Model-Based Approach. In *International Conference on Computer Safety, Reliability, and Security*, pages 69–85. Springer.

[Kolovos *et al.*(2015)] Kolovos, D. S., Rose, L. M., Garcia-Dominguez, A., and Paige, R. F. (2015). *The Epsilon Book*. Eclipse.

[Korel *et al.*(2003)] Korel, B., Singh, I., Tahat, L., and Vaysburg, B. (2003). Slicing of State-Based Models. In *International Conference on Software Maintenance*, pages 34–43. IEEE.

[Laibinis *et al.*(2015)] Laibinis, L., Troubitsyna, E., Prokhorova, Y., Iliasov, A., and Romanovsky, A. (2015). From Requirements Engineering to Safety Assurance: Refinement Approach. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, volume 9409 of *LNCS*, pages 201–216.

[Lallchandani and Mall(2011)] Lallchandani, J. T. and Mall, R. (2011). A Dynamic Slicing Technique for UML Architectural Models. *IEEE Transactions on Software Engineering*, **37**(6), 737–771.

[Lämmel(2008)] Lämmel, R. (2008). Google's MapReduce Programming Model – Revisited. *Science of Computer Programming*, **70**(1), 1–30.

[Lano and Rahimi(2010)] Lano, K. and Rahimi, S. K. (2010). Slicing of UML Models. In *International Conference on Software Technologies*, pages 259–262.

[Larrucea(2016)] Larrucea, X. (2016). Modelling and Certifying Safety for Cyber-Physical Systems: An Educational Experiment. In *Software Engineering and Advanced Applications*, pages 198–205.

[Larrucea *et al.*(2017)] Larrucea, X., Walker, A., and Colomo-Palacios, R. (2017). Supporting the Management of Reusable Automotive Software. *IEEE Software Journal*, **34**(3), 40–47.

[Lautieri *et al.*(2004)] Lautieri, S., Cooper, D., Jackson, D., and Cockram, T. (2004). Assurance Cases: How Assured Are You? In *International Conference on Dependable Systems and Networks*.

[Leveson(2011)] Leveson, N. (2011). *Engineering a safer world: Systems thinking applied to safety*. MIT press.

[Leveson(1995)] Leveson, N. G. (1995). Safety as a system property. *Communications of the ACM*, **38**(11), 146.

[Lewis(2009)] Lewis, R. (2009). Safety Case Development as an Information Modelling Problem. In *Safety-Critical Systems: Problems, Process and Practice*, pages 183–193.

[Li *et al.*(2013)] Li, B., Sun, X., Leung, H., and Zhang, S. (2013). A Survey of Code-Based Change Impact Analysis Techniques. *Journal of Software Testing, Verification and Reliability*, **23**(8), 613–646.

[Li(2016)] Li, Z. (2016). *A Systematic Approach and Tool Support for Assessing GSN-Based Safety Case*. Master's thesis, Technische Universiteit Eindhoven.

[Lucrédio *et al.*(2008)] Lucrédio, D., Fortes, R. P. d. M., and Whittle, J. (2008). MOOGLE: A Model Search Engine. In *International Conference on Model Driven Engineering Languages and Systems*, pages 296–310. Springer.

[Ludewig(2004)] Ludewig, J. (2004). Models in software engineering - an introduction. *Inform., Forsch. Entwickl.*, **18**(3-4), 105–112.

[Luo *et al.*(2013)] Luo, Y., van den Brand, M., Engelen, L., Favaro, J., Klabbers, M., and Sartori, G. (2013). Extracting Models from ISO 26262 for Reusable Safety Assurance. In *International Conference on Software Reuse*, pages 192–207. Springer.

[Luo *et al.*(2014)] Luo, Y., van den Brand, M., Engelen, L., and Klabbers, M. (2014). From conceptual models to safety assurance. In *International Conference on Conceptual Modeling*, pages 195–208. Springer.

[Luo *et al.*(2015a)] Luo, Y., van den Brand, M., Engelen, L., and Klabbers, M. (2015a). A Modeling Approach to Support Safety Assurance in the Automotive Domain. In *Progress in Systems Engineering*, pages 339–345.

[Luo *et al.*(2015b)] Luo, Y., van den Brand, M., and Kiburse, A. (2015b). Safety Case Development with SBVR-Based Controlled Language. In *International Conference on Model-Driven Engineering and Software Development*, volume 580, pages 3–17.

[Luo *et al.*(2016)] Luo, Y., Li, Z., and van den Brand, M. (2016). A categorization of gsn-based safety cases and patterns. In *Model-Driven Engineering and Software Development (MODELSWARD), 2016 4th International Conference on*, pages 509–516. IEEE.

[Luo *et al.*(2017)] Luo, Y., van den Brand, M., Li, Z., and Saberi, A. (2017). A Systematic Approach and Tool Support for GSN-Based Safety Case Assessment. *Journal of Systems Architecture*, **76**(pp), 1 – 16.

[Maksimov *et al.*(2018)] Maksimov, M., Fung, N. L. S., Kokaly, S., and Chechik, M. (2018). Two Decades of Assurance Case Tools: A Survey. In *International Conference on Computer Safety, Reliability, and Security*. Springer. accepted for publication.

[Matsuno(2017)] Matsuno, Y. (2017). D-Case Communicator: A Web Based GSN Editor for Multiple Stakeholders. In *International Conference on Computer Safety, Reliability, and Security*, volume 10489 of *LNCS*, pages 64 – 69.

[Matsuno *et al.*(2010)] Matsuno, Y., Takamura, H., and Ishikawa, Y. (2010). A Dependability Case Editor with Pattern Library. In *High-Assurance Systems Engineering*, pages 170–171.

[Mellor *et al.*(2004)] Mellor, S. J., Kendall, S., Uhl, A., and Weise, D. (2004). *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Melnik *et al.*(2003)] Melnik, S., Rahm, E., and Bernstein, P. A. (2003). Rondo: A programming platform for generic model management. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 193–204. ACM.

[Millett *et al.*(2007)] Millett, L. I., Thomas, M., Jackson, D., *et al.* (2007). *Software for Dependable Systems:: Sufficient Evidence?* National Academies Press.

[Nair *et al.*(2015a)] Nair, S., Walkinshaw, N., Kelly, T., and de la Vara, J. L. (2015a). An Evidential Reasoning Approach for Assessing Confidence in Safety Evidence. In *International Symposium on Software Reliability Engineering*, pages 541–552.

[Nair *et al.*(2015b)] Nair, S., de la Vara, J. L., Sabetzadeh, M., and Falessi, D. (2015b). Evidence Management for Compliance of Critical Systems with Safety Standards: A Survey on the State of Practice. *Information and Software Technology*, **60**, 1–15.

[Nejati *et al.*(2012)] Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., and Coq, T. (2012). A SysML-based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies. *Information and Software Technology*, **54**(6), 569–590.

[Netkachova *et al.*(2015)] Netkachova, K., Netkachov, O., and Bloomfield, R. (2015). Tool Support for Assurance Case Building Blocks. In *International Conference on Computer Safety, Reliability, and Security*, volume 9338 of *LNCS*, pages 62–71.

[Newton and Vickers(2007)] Newton, A. and Vickers, A. (2007). The Benefits of Electronic Safety Cases. In *Safety-Critical Systems: Problems, Process and Practice*, pages 69–82.

[Noda *et al.*(2009)] Noda, K., Kobayashi, T., Agusa, K., and Yamamoto, S. (2009). Sequence Diagram Slicing. In *Asia-Pacific Software Engineering Conference*, pages 291–298. IEEE.

[Object Management Group(2014)] Object Management Group (2014). *Model Driven Architecture (MDA) MDA Guide rev. 2.0.*

[Object Management Group(2015a)] Object Management Group (2015a). *OMG Unified Modeling Language TM (OMG UML) Version 2.5.*

[Object Management Group(2015b)] Object Management Group (2015b). *XML Metadata Interchange (XMI) Specification, Version 2.5.1.*

[OMG(2015)] OMG (2015). OMG's MetaObject Facility. http://www.omg.org/mof/.

[OMG(2015)] OMG (2015). Structured Assurance Case Metamodel (SACM). http://www.omg.org/spec/SACM/.

[OMG(2016)] OMG (2016). Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.

[Paige *et al.*(2016)] Paige, R. F., Matragkas, N., and Rose, L. M. (2016). Evolving Models in Model-Driven Engineering: State-of-the-art and Future Challenges. *Journal of Systems and Software*, **111**, 272–280.

[Palin *et al.*(2011)] Palin, R., Ward, D., Habli, I., and Rivett, R. (2011). ISO 26262 Safety Cases: Compliance and Assurance.

[Panesar-Walawege *et al.*(2013)] Panesar-Walawege, R. K., Sabetzadeh, M., and Briand, L. (2013). Supporting the verification of compliance to safety standards via model-driven engineering: Approach, tool-support and empirical validation. *Information and Software Technology*, **55**(5), 836–864.

[PREEVision(2018)] PREEVision (2018). PREEVision. https://vector.com/vi_preevision-iso26262_en.html.

[Pro(2018)] Pro, S. (2018). SMS Pro. https://www.asms-pro.com/Modules/SafetyAssurance/SafetyCaseStudy.aspx.

[Ratiu *et al.*(2015)] Ratiu, D., Zeller, M., and Killian, L. (2015). Safety.Lab: Model-Based Domain Specific Tooling for Safety Argumentation. In *International Conference on Computer Safety, Reliability, and Security*, volume 9338 of *LNCS*, pages 72–82.

[Retouniotis *et al.*(2017)] Retouniotis, A., Papadopoulos, Y., Sorokos, I., Parker, D., Matragkas, N., and Sharvia, S. (2017). Model-Connected Safety Cases. In *International Symposium on Model-Based Safety and Assessment*, volume 10437 of *LNCS*, pages 50–63.

[Sabetzadeh *et al.*(2013)] Sabetzadeh, M., Falessi, D., Briand, L., and Di Alesio, S. (2013). A Goal-Based Approach for Qualification of New Technologies: Foundations, Tool Support, and Industrial Validation. *Reliability Engineering & System Safety*, **119**(C), 52 – 66.

[Salay *et al.*(2007)] Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., and Viriyakattiyaporn, P. (2007). An Eclipse-Based Tool Framework for Software Model Management. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 55–59.

[Salay *et al.*(2009)] Salay, R., Mylopoulos, J., and Easterbrook, S. (2009). Using Macromodels to Manage Collections of Related Models. In *International Conference on Advanced Information Systems Engineering*, pages 141–155. Springer.

[Salay *et al.*(2014)] Salay, R., Famelis, M., Rubin, J., Di Sandro, A., and Chechik, M. (2014). Lifting Model Transformations to Product Lines. In *Proceedings of the 36th International Conference on Software Engineering*, pages 117–128. ACM.

[Salay *et al.*(2015)] Salay, R., Kokaly, S., Di Sandro, A., and Chechik, M. (2015). Enriching Megamodel Management with Collection-Based Operators. In *International Conference on Model Driven Engineering Languages and Systems*, pages 236–245. IEEE.

[Salay *et al.*(2016)] Salay, R., Kokaly, S., Chechik, M., and Maibaum, T. (2016). Heterogeneous Megamodel Slicing for Model Evolution. In *Models and Evolution Workshop at Model Driven Engineering Languages and Systems (MODELS)*, pages 50–59.

[Shida *et al.*(2013)] Shida, S., Uchida, A., Ishii, M., Ide, M., and Kuramitsu, K. (2013). Assure-It: A Runtime Synchronization Tool of Assurance Cases. In *International Conference on Computer Safety, Reliability, and Security*.

[Stahl *et al.*(2006)] Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

[Steinberg *et al.*(2008)] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

[Stürmer *et al.*(2012)] Stürmer, I., Salecker, E., and Pohlheim, H. (2012). Reviewing software models in compliance with iso 26262. In *International Conference on Computer Safety, Reliability, and Security*, pages 258–267. Springer.

[The GSN Working Group(2015)] The GSN Working Group (2015). SACM 2.0 Argumentation Example. http://www.goalstructuringnotation.info/wp-content/uploads/2015/11/SACM-2-examples.pdf.

[TurboAC(2018)] TurboAC (2018). TurboAC. http://www.gessnet.com/products.

[Ujhelyi *et al.*(2011)] Ujhelyi, Z., Horváth, Á., and Varró, D. (2011). Towards dynamic backward slicing of model transformations. In *26th IEEE/ACM International Conference on Automated Software Engineering*, pages 404–407. IEEE.

[Ujhelyi *et al.*(2015)] Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. (2015). EMF-IncQuery: An Integrated Development Environment for Live Model Queries. *Science of Computer Programming*, **98**, 80–99.

[University of York(2015)] University of York (2015). D4.2 Compositional Assurance Cases and Arguments for Distributed MILS.

[University of York(2018)] University of York (2018). Impact case study - University of York. https://impact.ref.ac.uk/CaseStudies/CaseStudy.aspx?Id=43445.

[Vanhooff *et al.*(2007)] Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., and Berbers, Y. (2007). Uniti: A Unified Transformation Infrastructure. In *International Conference on Model Driven Engineering Languages and Systems*, pages 31–45. Springer.

[Vignaga *et al.*(2013)] Vignaga, A., Jouault, F., Bastarrica, M. C., and Brunelière, H. (2013). Typing Artifacts in Megamodeling. *Journal of Software and Systems Modeling*, **12**(1), 105–119.

[Weiser(1981)] Weiser, M. (1981). Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press.

[Widl *et al.*(2012)] Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., and Tompits, H. (2012). Guided Merging of Sequence Diagrams. In *International Conference on Software Language Engineering*, pages 164–183. Springer.

[Zhang *et al.*(2011)] Zhang, H., Babar, M. A., and Tell, P. (2011). Identifying Relevant Studies in Software Engineering. *Journal of Information and Software Technology*, **53**(6), 625–637.

Appendix A

Power Sliding Door Models

This appendix presents all the Power Sliding Door (PSD) system design models encoded in *MMINT*.

A.1 PSD Class Diagram

Figure A.1 displays a class diagram for the PSD encoded in our tool *MMINT*. This class diagram describes the objects involved in the PSD and their relationships with one another.

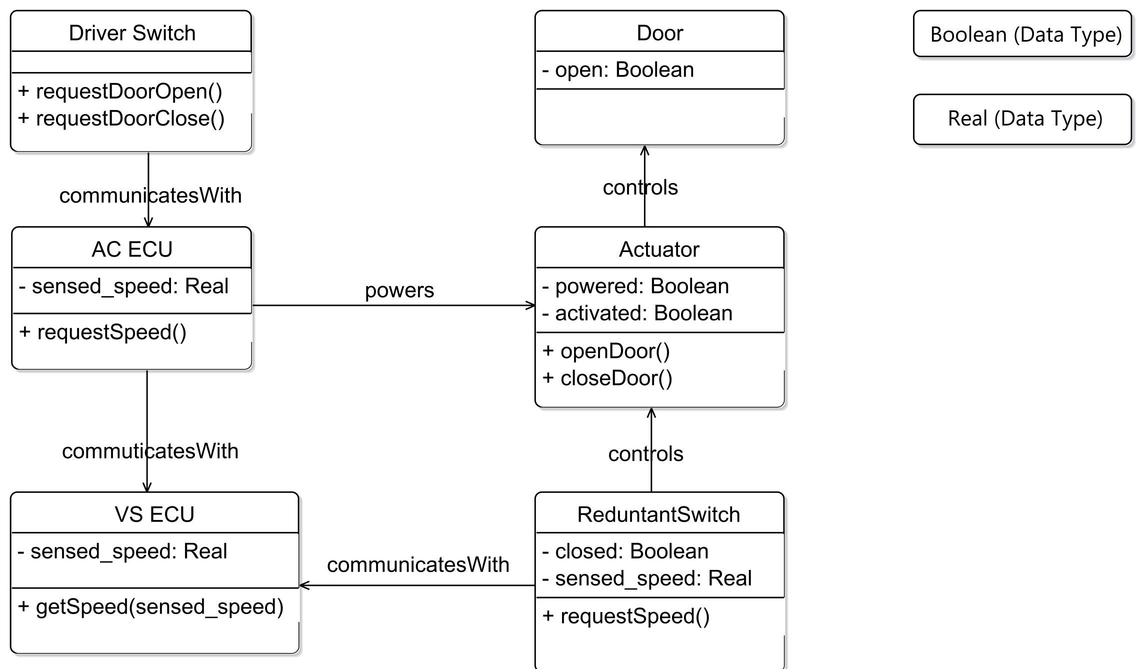


Figure A.1: Power Sliding Door (PSD) Class Diagram

A.2 PSD Sequence Diagram

Figure A.2 displays a sequence diagram for the PSD encoded in our tool *MMINT*. Sequence diagrams are used to emphasize the interactions between objects within a system. Sequence diagrams have a series of boxes at the top of the image that depicts objects. From these boxes are dashed lines. Between the dashed lines are synchronous messages and responses that occur between the objects.

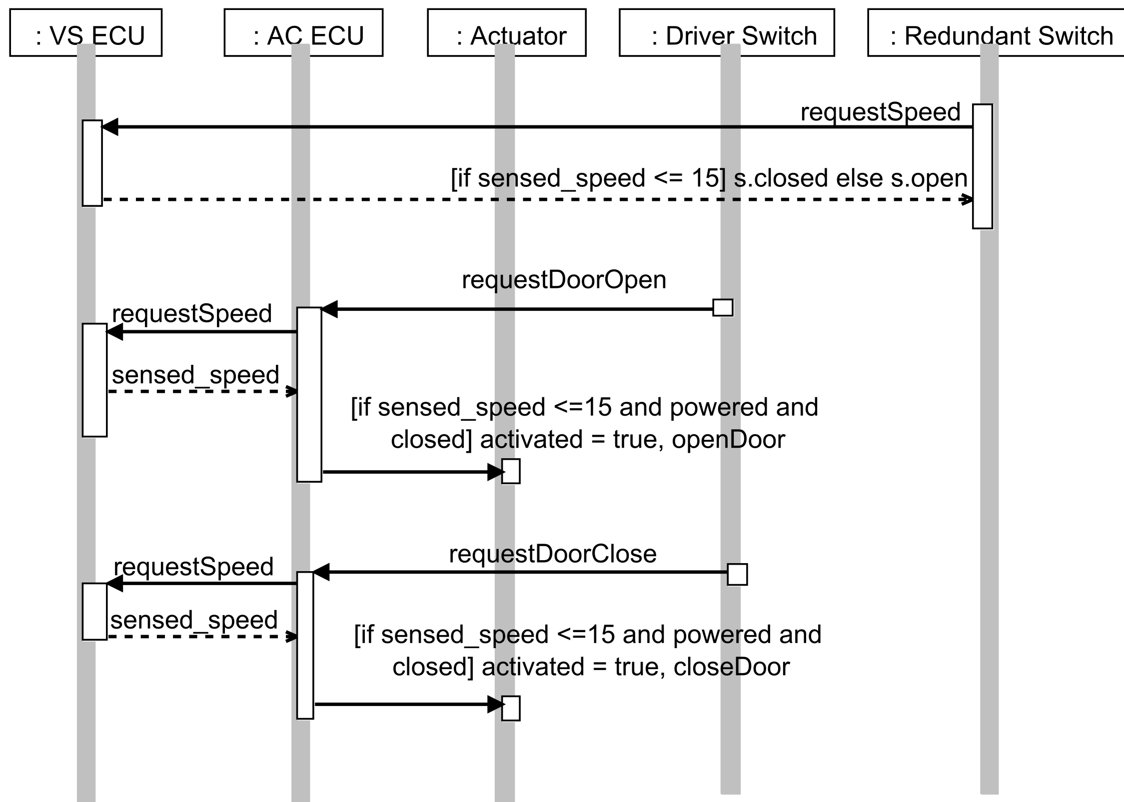


Figure A.2: Power Sliding Door (PSD) Sequence Diagram

Appendix B

Lane Management System Models

This appendix presents all the Lane Management System (LMS) design models encoded in *MMINT* and traceability relationships between them.

B.1 LMS Class Diagram

Figure B.3 displays a class diagram for the LMS taken from [Blazy *et al.*(2014)] and encoded in our tool *MMINT*. This class diagram describes the objects involved in the LMS and their relationships with one another. The class diagram consists of boxes that depict classes, which hold relative attributes and operations within them. In addition, associations (relationships) between the classes are depicted with a line, as well as a description and arrow. Within Figure B.3, there are also special associations called aggregations depicted via a diamond. An aggregation is used for showing when a class is a part of another class. For more information on the system, including a data dictionary, please refer to [Blazy *et al.*(2014)].

B.2 LMS Sequence Diagrams

Below are the representative scenarios of the LMS encoded using sequence diagrams for each scenario taken from [Blazy *et al.*(2014)] and encoded in *MMINT*. Sequence diagrams are used to emphasize the interactions between objects within a system. Sequence diagrams have a series of boxes at the top of the image that depicts objects. From these boxes are dashed lines. Between the dashed lines are synchronous messages and responses that occur between the objects.

B.2.1 LMS DrivingStraight Sequence Diagram

Figure B.4 depicts the DrivingStraight Scenario. This scenario depicts how the system handles driving straight. For example, with this scenario it is assumed that the vehicle will be traveling in a

straight line with zero road curvature, valid and conforming sequence of images from the subsystems and vehicle speed is uniform.

B.2.2 LMS FailureState Sequence Diagram

Figure B.5 depicts the FailureState Scenario. This scenario depicts how the system handles failure. For example, lane information may be missing or a subsystem may have failed.

B.2.3 LMS LeftCurve Sequence Diagram

Figure B.6 depicts the LeftCurve Scenario. This scenario depicts how the system handles non-zero curvature.

B.2.4 LMS SystemOn Sequence Diagram

Figure B.7 depicts the SystemOn Scenario. This scenario depicts how the system handles being on, with no communication between the system and the driver or the system and the systems subsystems.

B.3 LMS State Diagrams

This section contains depictions of the respective state diagrams for the LMS and its subsystems. A state diagram portrays the behavior of a system. It is made up of states (circles) and events (lines between states). Figures B.8, B.9, B.10 and B.11 depict the overall LMS state diagram, the LKS state diagram, the LCS state diagram and the LDWS state diagram, respectively.

B.4 Traceability between LMS models

Figures B.12 to B.19 show in a table format the traceability relations between the various LMS system models.

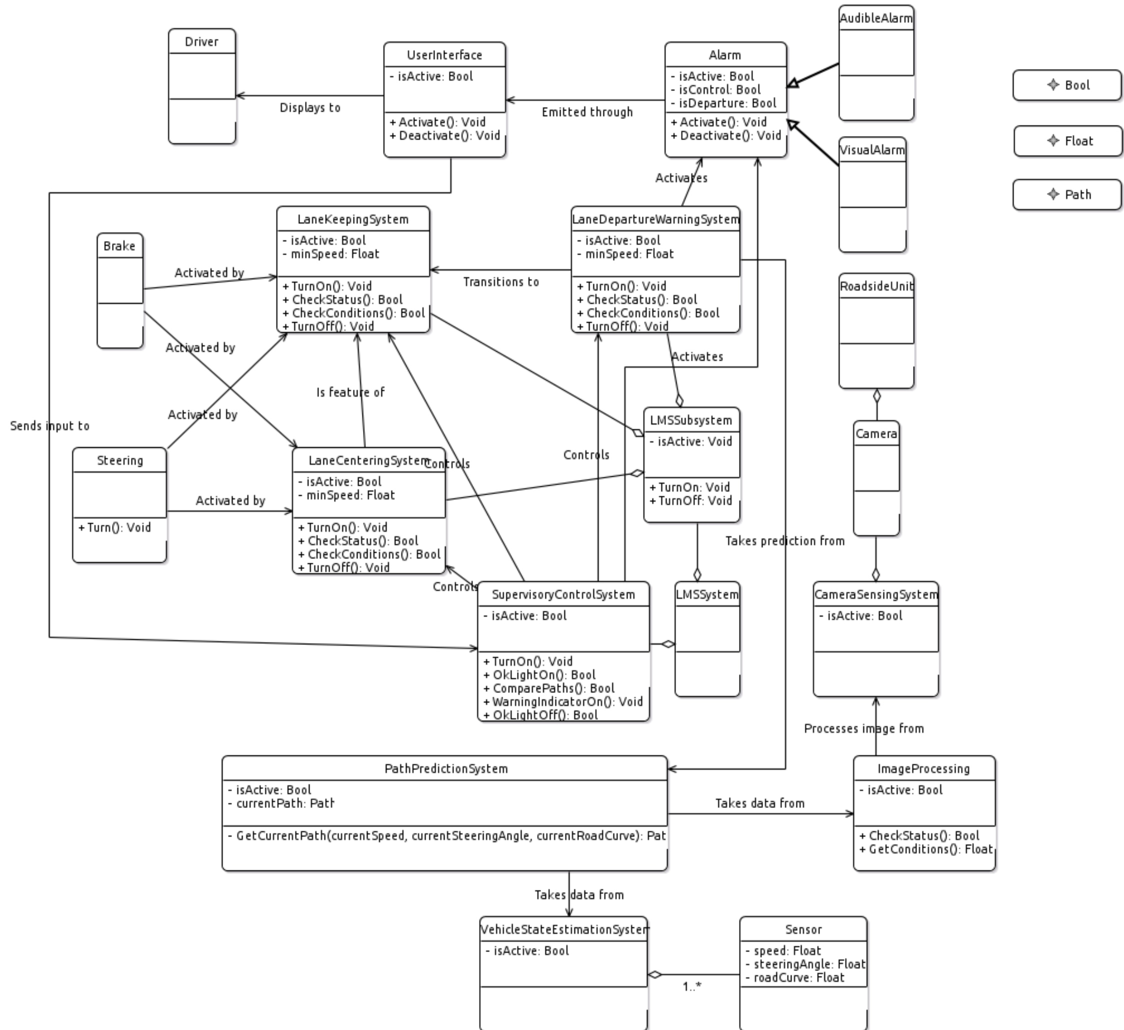


Figure B.3: Lane Management System (LMS) Class Diagram

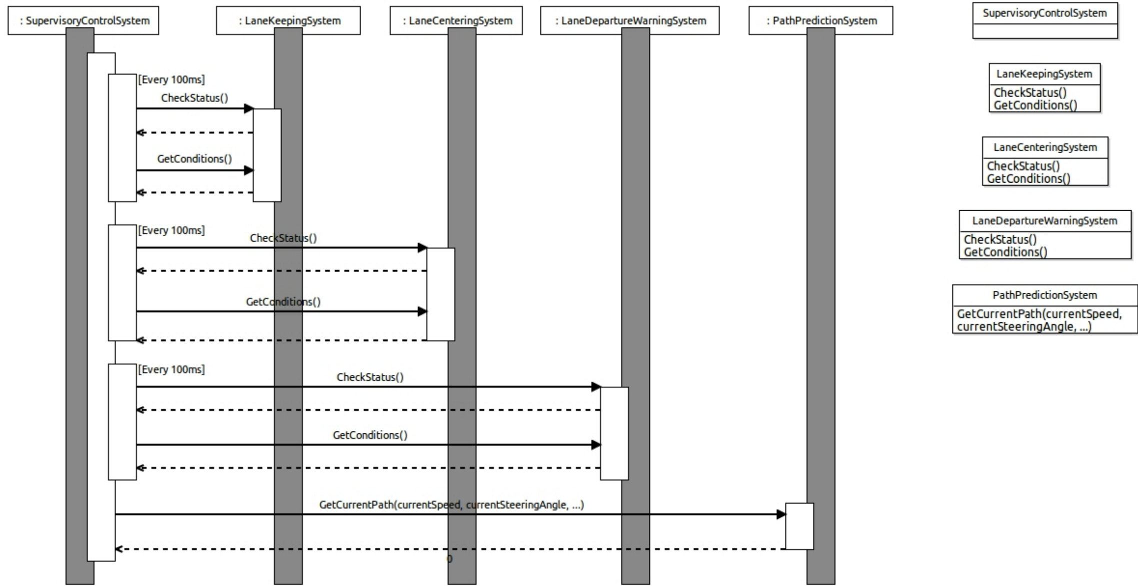


Figure B.4: Lane Management System (LMS) Sequence Diagram - DrivingStraight

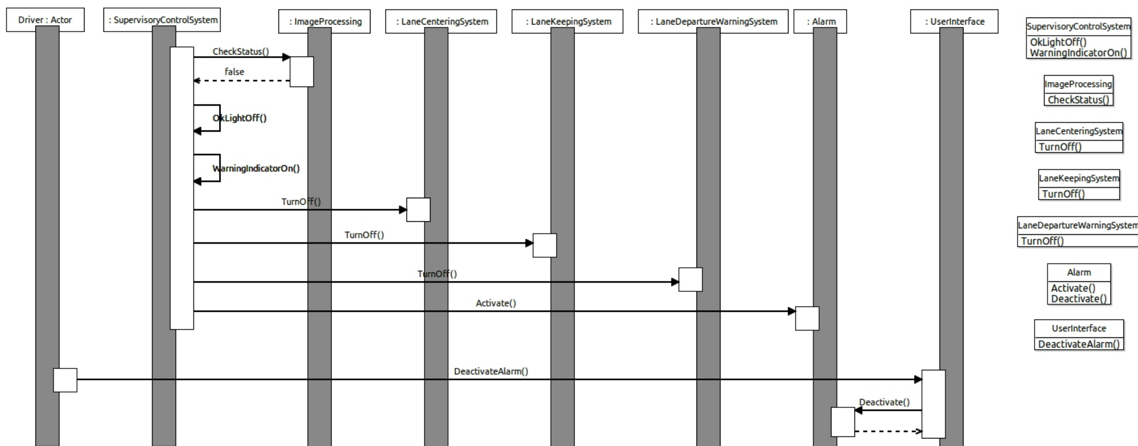


Figure B.5: Lane Management System (LMS) Sequence Diagram - FailureState

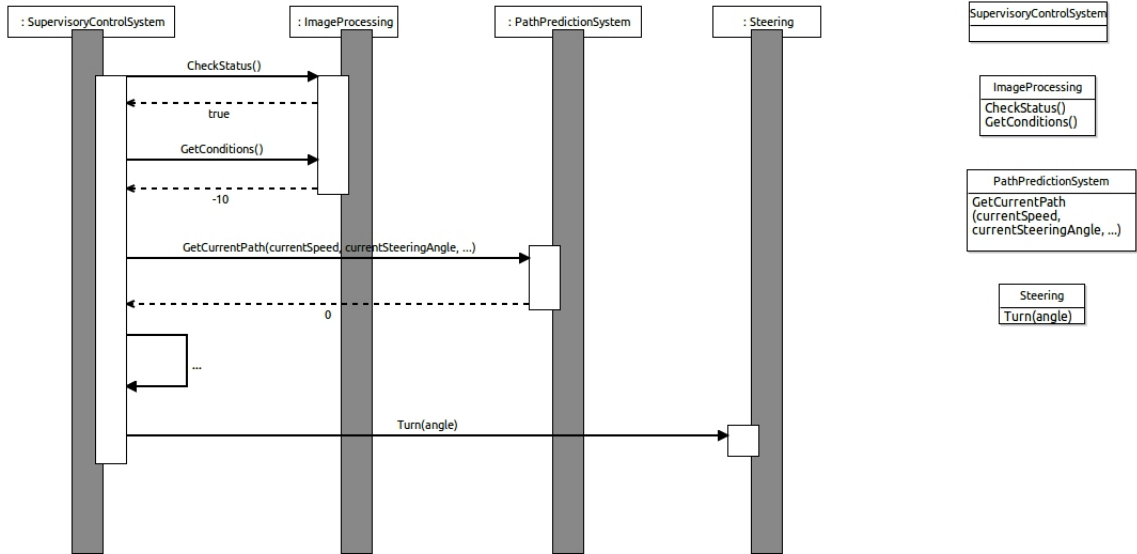


Figure B.6: Lane Management System (LMS) Sequence Diagram - LeftCurve

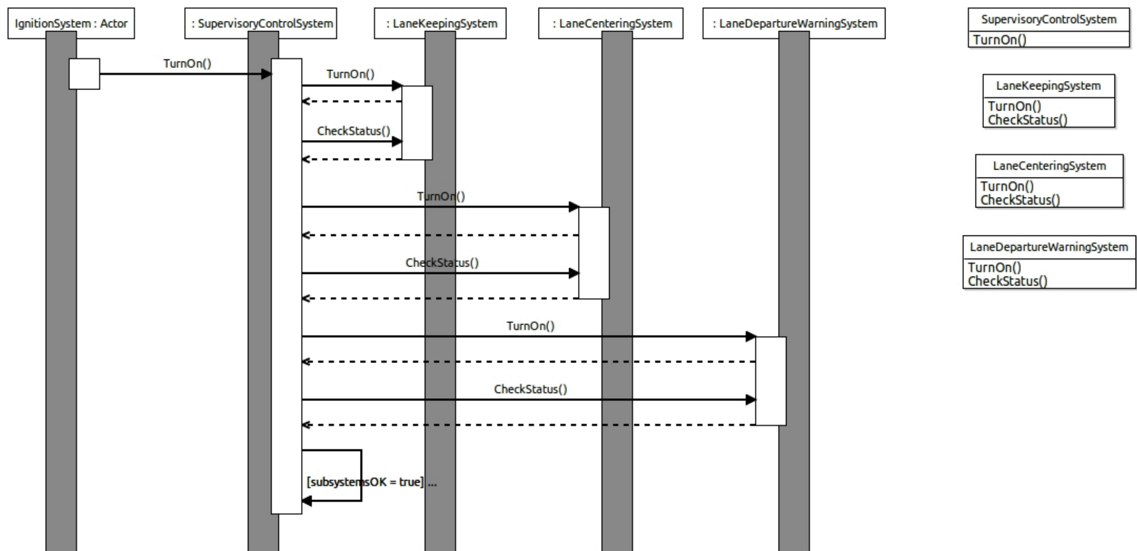


Figure B.7: Lane Management System (LMS) Sequence Diagram - SystemOn

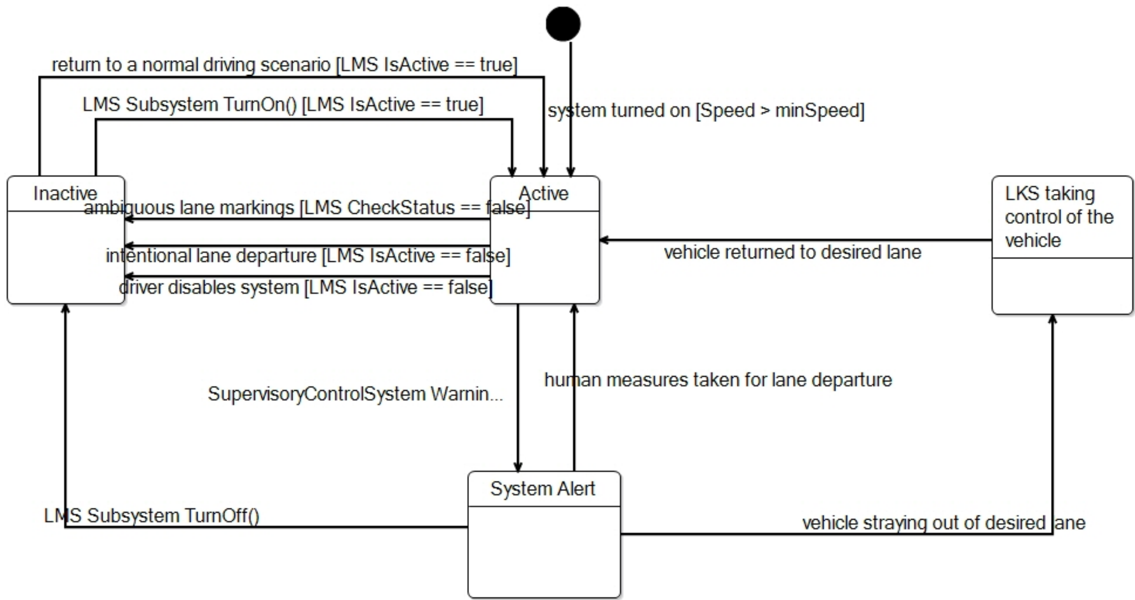


Figure B.8: LMS State Diagram

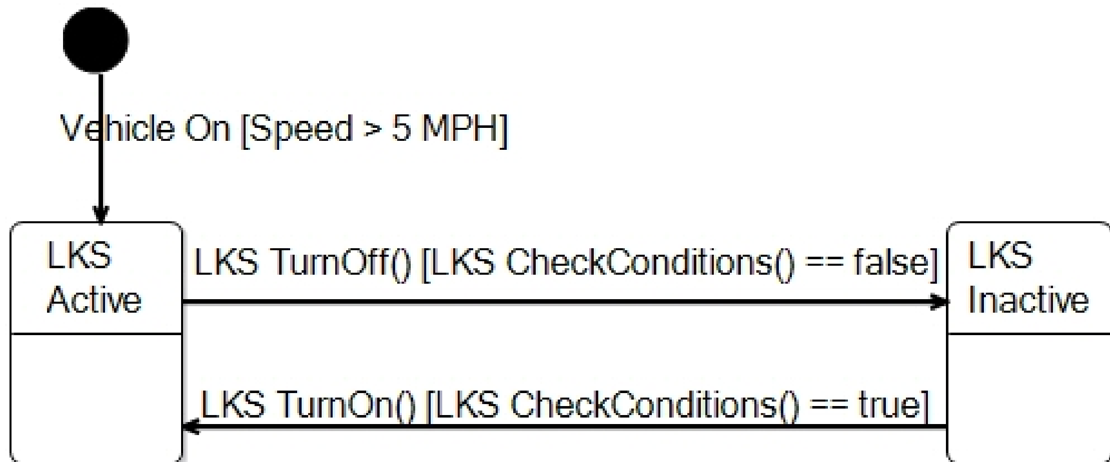


Figure B.9: LKS State Diagram

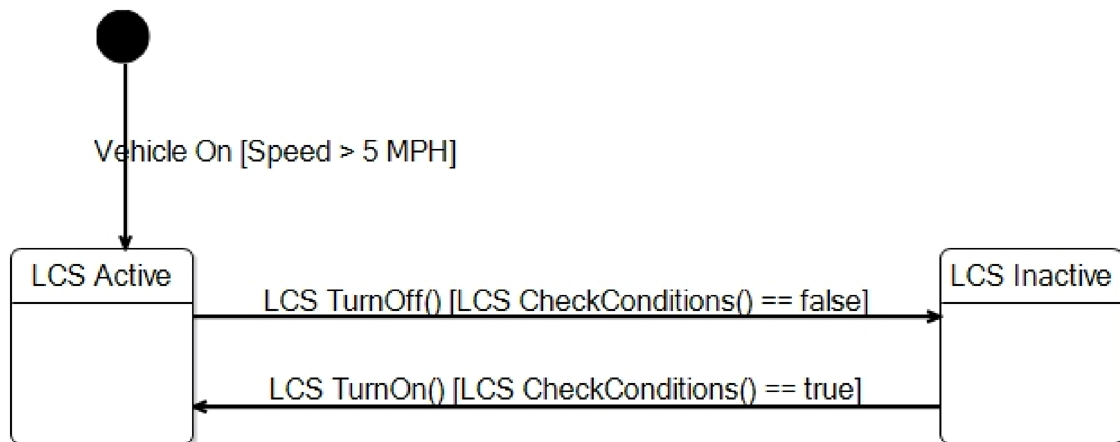


Figure B.10: LCS State Diagram

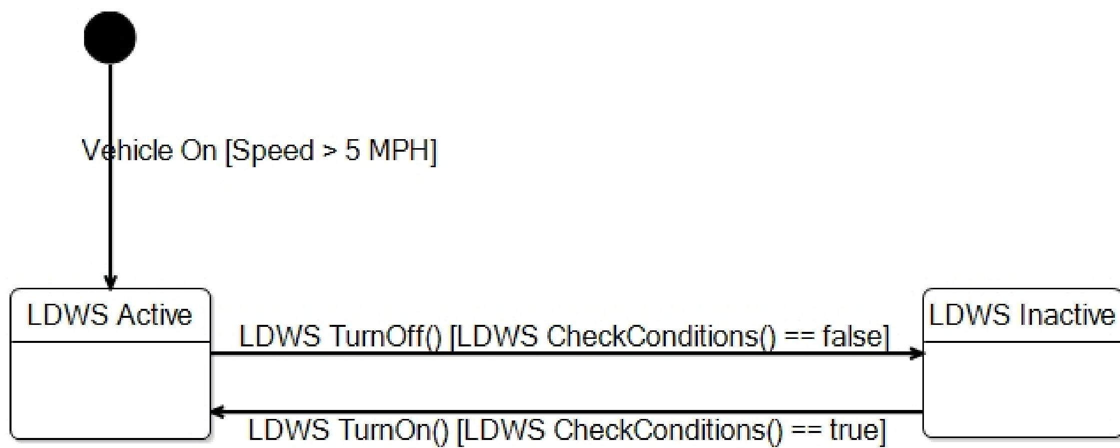


Figure B.11: LDWS State Diagram

			From: Class Diagram Element								
			Class	Class	Operation	Class	Operation	Operation	Class	Class	Class
			LMSSystem	LaneKeeping System	TurnOn	Steering	Warning IndicatorOn()	TurnOff	Image Processing	User Interface	PathPrediction System
					(Class LMS Subsystem)		(Class SCS)	(Class LMS Subsystem)			
To: Sequence Diagram Element	**State**	Active	√								
	State	System Alert	√								
	State	Inactive	√								
	State	LKS taking control of the vehicle		√							
	Transition	system turned on			√						
	Transition	vehicle returned to desired lane				√					
	Transition	human measures taken for lane departure				√					
	Transition	vehicle straying out of desired lane				√					
	Transition	SupervisoryControlSystem WarningIndicatorOn()					√				
	Transition	LMS Subsystem TurnOff()						√			
	Transition	intentional lane departure				√					
	Transition	ambiguous lane markings							√		
	Transition	driver disables system								√	
	Transition	return to a normal driving scenario									√
	Transition	LMS Subsystem TurnOn()	√								

Figure B.12: Traceability between LMS CD and LMS State Diagram

			From: Class Diagram Element		
			Class	Operation	Operation
			LaneKeeping System	TurnOn()	TurnOff()
				(Class LKS)	(Class LKS)
To: Sequence Diagram Element	**State**	LKS Active	√		
	State	LKS Inactive	√		
	Transition	Vehicle On		√	
	Transition	LKS TurnOff()			√
	Transition	LKS TurnOn()		√	

Figure B.13: Traceability between LMS CD and LKS State Diagram

			From: Class Diagram Element		
			Class	Operation	Operation
			LaneDeparture WarningSystem	TurnOn()	TurnOff()
				(Class LDWS)	(Class LDWS)
To: Sequence Diagram Element	State	LDWS Active	√		
	State	LDWS Inactive	√		
	Transition	Vehicle On		√	
	Transition	LDWS TurnOff()			√
	Transition	LDWS TurnOn()		√	

Figure B.14: Traceability between LMS CD and LDWS State Diagram

			Class	Operation	Operation
			LaneCentering System	TurnOn()	TurnOff()
				(Class LCS)	(Class LCS)
To: Sequence Diagram Element	State	LCS Active	√		
	State	LCS Inactive	√		
	Transition	Vehicle On		√	
	Transition	LCS TurnOff()			√
	Transition	LCS TurnOn()		√	

Figure B.15: Traceability between LMS CD and LCS State Diagram

				From: Class Diagram Element										
				Class	Class	Class	Class	Operation	Operation	Operation	Operation	Operation	Operation	Operation
				Supervisory ControlSystem	LaneKeeping System	LaneCentering System	LaneDeparture WarningSystem	TurnOn()	TurnOn()	CheckStatus()	TurnOn()	CheckStatus()	TurnOn()	CheckStatus()
								(Class SCS)	(Class LKS)	(Class LKS)	(Class LCS)	(Class LCS)	(Class LDWS)	(Class LDWS)
To: Sequence Diagram Element	Component	Supervisory ControlSystem		√										
	Component	LaneKeeping System			√									
	Component	LaneCentering System				√								
	Component	LaneDeparture WarningSystem					√							
	Message	TurnOn()	(Ignition -> SCS)					√						
	Message	TurnOn()	(SCS -> LKS)						√					
	Message	CheckStatus()	(SCS -> LKS)							√				
	Message	TurnOn()	(SCS -> LCS)								√			
	Message	CheckStatus()	(SCS -> LCS)									√		
	Message	TurnOn()	(SCS -> LDWS)										√	
	Message	CheckStatus()	(SCS -> LDWS)											√

Figure B.16: Traceability between LMS CD and LMS SystemOn Sequence Diagram

				From: Class Diagram Element							
				Class	Class	Class	Class	Operation	Operation	Operation	Operation
				Supervisory ControlSystem	Image Processing	PathPrediction System	Steering	Check Status()	GetConditions()	GetCurrent Path(...)	Turn()
								(Class IP)	(Class IP)	(Class PPS)	(Class Steering)
To: Sequence Diagram Element	**Component**	Supervisory ControlSystem		√							
	Component	Image Processing			√						
	Component	PathPrediction System				√					
	Component	Steering					√				
	Message	CheckStatus()	(SCS -> IP)					√			
	Message	GetConditions()	(SCS -> IP)						√		
	Message	GetCurrentPath(...)	(SCS -> PPS)							√	
	Message	Turn(-10)	(SCS -> Steering)								√

Figure B.17: Traceability between LMS CD and LMS LeftCurve Sequence Diagram

				From: Class Diagram Element															
				Class	Class	Class	Class	Class	Class	Class	Operation	Operation	Operation	Operation	Operation	Operation	Operation	Operation	Operation
				Supervisory ControlSystem	Image Processing	LaneCentering System	LaneKeeping System	LaneDeparture WarningSystem	Alarm	User Interface	CheckStatus()	OkLightOff()	Warning IndicatorOn()	TurnOff()	TurnOff()	TurnOff()	Activate()	Deactivate()	Deactivate()
											(Class IP)	(Class SCS)	(Class SCS)	(Class LCS)	(Class LKS)	(Class LDWS)	(Class Alarm)	(Class UI)	(Class Alarm)
To: Sequence Diagram Element	**Component**	Supervisory ControlSystem		√															
	Component	Image Processing			√														
	Component	LaneCentering System				√													
	Component	LaneKeeping System					√												
	Component	LaneDeparture WarningSystem						√											
	Component	Alarm							√										
	Component	UserInterface								√									
	Message	CheckStatus()	(SCS -> IP)								√								
	Message	OkLightOff()	(SCS -> SCS)									√							
	Message	Warning IndicatorOn()	(SCS -> SCS)										√						
	Message	TurnOff()	(SCS -> LCS)											√					
	Message	TurnOff()	(SCS -> LKS)												√				
	Message	TurnOff()	(SCS -> LDWS)													√			
	Message	Activate()	(SCS -> Alarm)														√		
	Message	Deactivate Alarm()	(Driver -> UI)															√	
	Message	Deactivate()	(UI -> Alarm)																√

Figure B.18: Traceability between LMS CD and LMS FailureState Sequence Diagram

				From: Class Diagram Element											
				Class	Class	Class	Class	Class	Operation	Operation	Operation	Operation	Operation	Operation	Operation
				Supervisory ControlSystem	LaneKeeping System	LaneCentering System	LaneDeparture WarningSystem	PathPredic- tionSystem	Check Status()	Check Conditions()	Check Status()	Check Conditions()	Check Status()	Check Conditions()	GetCurrentPath(currentSpeed,...)
									(Class LKS)	(Class LKS)	(Class LCS)	(Class LCS)	(Class LDWS)	(Class LDWS)	(Class PPS)
To: Sequence Diagram Element	**Component**	Supervisory ControlSystem		√											
	Component	LaneKeeping System			√										
	Component	LaneCentering System				√									
	Component	LaneDeparture WarningSystem					√								
	Component	PathPredic- tionSystem						√							
	Message	CheckStatus()	(SCS -> LKS)						√						
	Message	GetConditions()	(SCS -> LKS)							√					
	Message	CheckStatus()	(SCS -> LCS)								√				
	Message	GetConditions()	(SCS -> LCS)									√			
	Message	CheckStatus()	(SCS -> LDWS)										√		
	Message	GetConditions()	(SCS -> LDWS)											√	
	Message	GetCurrentPath(...)	(SCS -> PPS)												√

Figure B.19: Traceability between LMS CD and LMS DrivingStraight Sequence Diagram

Appendix C

MMINT-A User Manual

This appendix contains a user manual for *MMINT* and *MMINT-A*, which can also be found online at: https://github.com/nlsfung/MMINT/wiki.

MMINT/MMINT-A User Manual

Home

Welcome to the MMINT wiki, which contains instructions on how to use MMINT and MMINT-A. In particular, this wiki is written based on personal experience of what works and, where appropriate, what doesn't. As a consequence, the user may find that there are undocumented but equally valid alternatives for completing each task, especially if he or she is already familiar with Eclipse, EMF and Sirius. Furthermore, familiarity with terms and concepts related to model management, MMINT and MMINT-A are assumed; background information on MMINT can be found on this wiki (https://github.com/adisandro/MMINT/wiki).

As shown in the sidebar, this wiki comprises four chapters:

1. **Installing MMINT** contains instructions for installing MMINT and Eclipse.
2. **Using MMINT** documents the features of MMINT and how to use them.
3. **Using MMINT-A** contains instructions on setting up and using MMINT-A, which is a set of extensions on top of MMINT for performaing change impact assessment on assurance cases.
4. **Extending MMINT** documents how to incorporate personal extensions into MMINT, such as new metamodels and new model operators.

Both **Using MMINT-A** and **Extending MMINT** assume some familiarity with using MMINT and thereby depend on the chapter **Using MMINT** (which in turn assume successful installation of MMINT). However, these two chapters are independent of each other despite the chronological order, thus users who only wish to work with assurance cases need not be familiar with **Extending MMINT**. Similarly, those who wish to extend MMINT for other applications need not learn about **Using MMINT-A**.

1 Installing MMINT

Software Requirements

Below is a list of the prerequisites for running MMINT. Optional software are indicated as such with an explanation of the circumstances under which they are required or recommended.

- Java 10 JDK
- Git (Optional. For keeping MMINT up-to-date.)
- Eclipse Modeling Tools, Photon Release w/ the following extra add-ons (see the following section for instructions):

- Acceleo
- ATL SDK (Required for class diagrams)
- m2e - Maven Integration for Eclipse (Optional. For build automation.)
- Papyrus for UML
- Sirius Properties Views - Specifier Support (Optional. For using Sirius.)
- Sirius Specifier Environment
- Tycho Project Configurators (Optional. For build automation.)

Installing Eclipse

The following instructions assume that Eclipse Modeling Tools (and the extra add-ons) are to be installed from scratch. If a version of Eclipse Photon already exists, then the instructions may need to be modified accordingly depending on which add-ons are missing.

1. Download installer for Eclipse Photon (http://www.eclipse.org/photon/ (http://www.eclipse.org/photon/))
2. Execute installer and choose to install Eclipse Modeling Tools.
3. Open Eclipse. For now, any directory can serve as the workspace.
4. Ensure that Eclipse is using the Java 10.
   1. Select Window > Preferences > Java > Installed JREs at the menu bar.
   2. Make sure Java 10 is listed and seleted. If not, then do so.
5. Install the additional components in Eclipse.
   1. Select Help > Install New Software
   2. Choose to work with "Photon - http://download.eclipse.org/releases/photon (http://download.eclipse.org /releases/photon)"
   3. Search for, select and install the components listed above except Tycho Project Configurators, which is located in a different repository (see Fig. 1.1).
   4. Do not restart Eclipse. Instead repeat steps a-c again but:
      1. Add and use the following respository: http://repo1.maven.org/maven2/.m2e/connectors/m2eclipse-tycho/0.8.1/N/0.8.1.201704211436/ (http://repo1.maven.org/maven2/.m2e/connectors/m2eclipse-tycho/0.8.1/N/0.8.1.201704211436/)
      2. Select and install Tycho Project Configurators
   5. Close or restart Eclipse to complete the installation process.

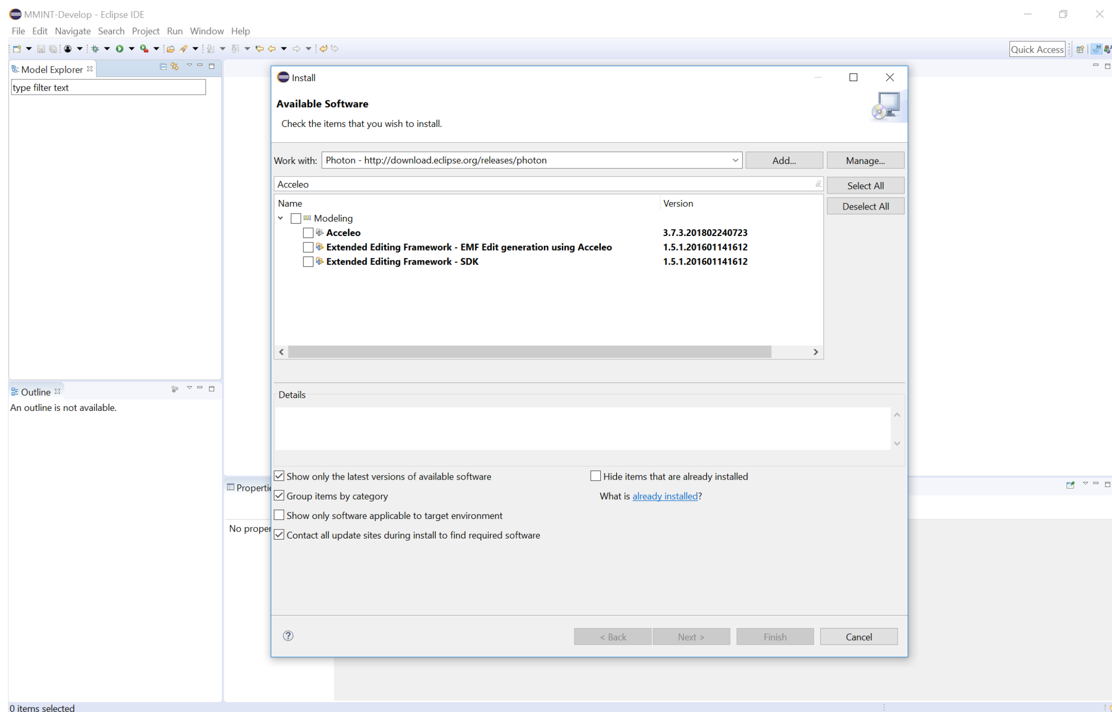


Fig. 1.1 A screenshot of the screen for installing new software in Eclipse. In this case, the target software is Acceleo, and it has been installed already.

Setting Up MMINT

The following instructions describe how to install MMINT and MMINT-A from source and assume familiarity with the use of Git (https://git-scm.com/) for version control. If the latest updates of MMINT-A are not required, then it is also possible to install MMINT and MMINT-A from binary packages, the instructions for which can be found at here (https://github.com/adisandro/MMINT/wiki/Install#binary-packages).

1. Go to a directory to contain the source files for MMINT (e.g. ~/Documents).

2. Clone one of the following Git repositories for MMINT
   - https://github.com/adisandro/MMINT.git (https://github.com/adisandro/MMINT.git) is the official repository

   - https://github.com/nlsfung/MMINT.git (https://github.com/nlsfung/MMINT.git) contains the most recent updates for MMINT-A.

3. Check out the "develop" branch of MMINT

4. Open Eclipse. Optionally, set the cloned MMINT directory to be the workspace.
   1. Select File > Workspace > Other... and look for the MMINT directory

5. Open the Java perspective.
   1. Select Window > Perspective in the menu bar, and look for Java

6. Import the projects for MMINT into Eclipse.

1. Select File > Import, then General > Existing Projects into Workspace

2. Choose the cloned MMINT directory as the root directory.

3. Under "Options", check "Search for nested projects".

4. Select all projects starting with "edu.toronto.cs.se" to import, except (see Fig. 1.2):

   - edu.toronto.cs.se.modelepedia.z3.operator

   - edu.toronto.cs.se.modelepedia.classdiagram_mavo.operator

   - edu.toronto.cs.se.modelepedia.icse14

After refreshing, Eclipse will report 10 errors that "plugin execution [is] not covered by lifecycle configuration". These errors do not have clear, observable impact on MMINT and can therefore be ignored.

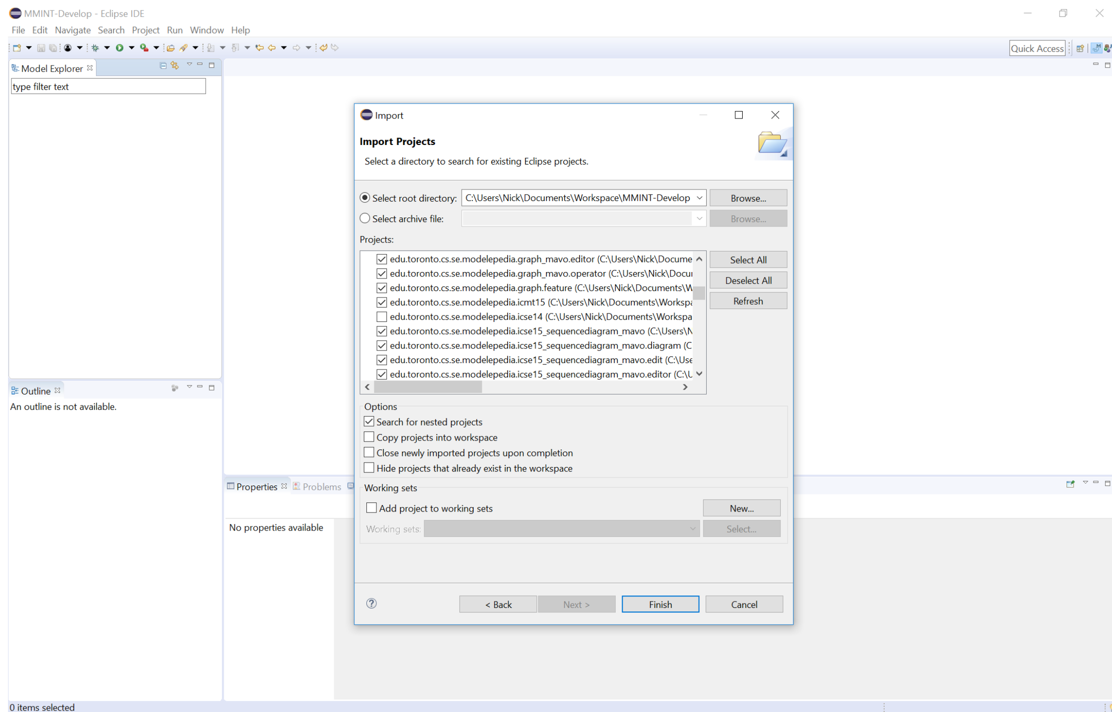


Fig. 1.2 A screenshot showing the projects to be imported.

2 Using MMINT

Contents

- Starting Up

- Configuring MMINT

- Inspecting the Type MID
- Creating a New Project
- Creating a Megamodel
- Using the MID Editor
- Using the ModelRel Editor
- Creating a Workflow
- Using the Workflow Editor
- Registering New Workflows

Starting Up

1. Run Eclipse. If necessary, switch to the workspace containing the MMINT source files.
2. Right-click on any MMINT project (e.g. the first one, edu.toronto.cs.se.mavo)
3. Select Debug As > Eclipse Application (see Fig. 2.1). If this option is not available, try another project.

Once these steps are completed, MMINT will launch as a new instance of Eclipse. By default, the workspace for MMINT is named "runtime-EclipseApplication" and is located under the same parent directory as the original workspace.

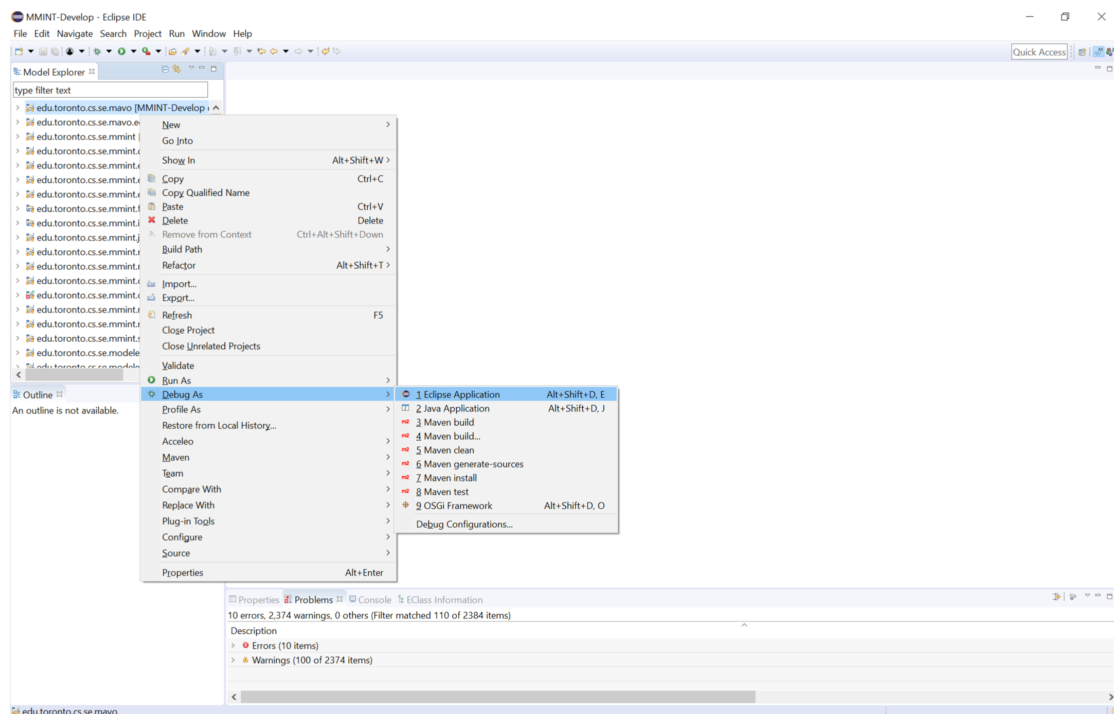


Fig. 2.1 A screenshot showing the final step in starting a new Eclipse application.

Configuring MMINT

The following configuration options are available in MMINT and can be enabled or disabled by opening up the MMINT menu in the menu bar and selecting or deselecting them (see Fig. 2.2).

- **Show icons on links** *Effects to be confirmed.*
- **Show labels on ModelRel links** Enables labels on newly-created binary links in a model relationship (see ModelRel Editor)
- **Show labels on ModelRel endpoints** Enables labels on the endpoints of newly-created n-ary relationships in a MID (see MID Editor) or in a ModelRel (see ModelRel Editor)
- **Show labels on Operator endpoints** *Effects to be confirmed*
- **Trace Operator execution** Keeps track of any operators used to create models in the MID. It is recommended that this option be enabled as it is required for some operators (e.g. workflows).
- **Open editors when Operators create Models** *Effects to be confirmed*
- **Delete Models from file system** Enables automatic deletion of models from the file system whenever they are deleted from a megamodel (i.e. an instance MID). Since this deletion is irreversible, it is recommendeded that this option be disabled.
- **Enable Runtime Typing** (Under Polymorphism sub-menu) *Effects to be confirmed*
- **Enable Multiple Dispatch** (Under Polymorphism sub-menu) *Effects to be confirmed*

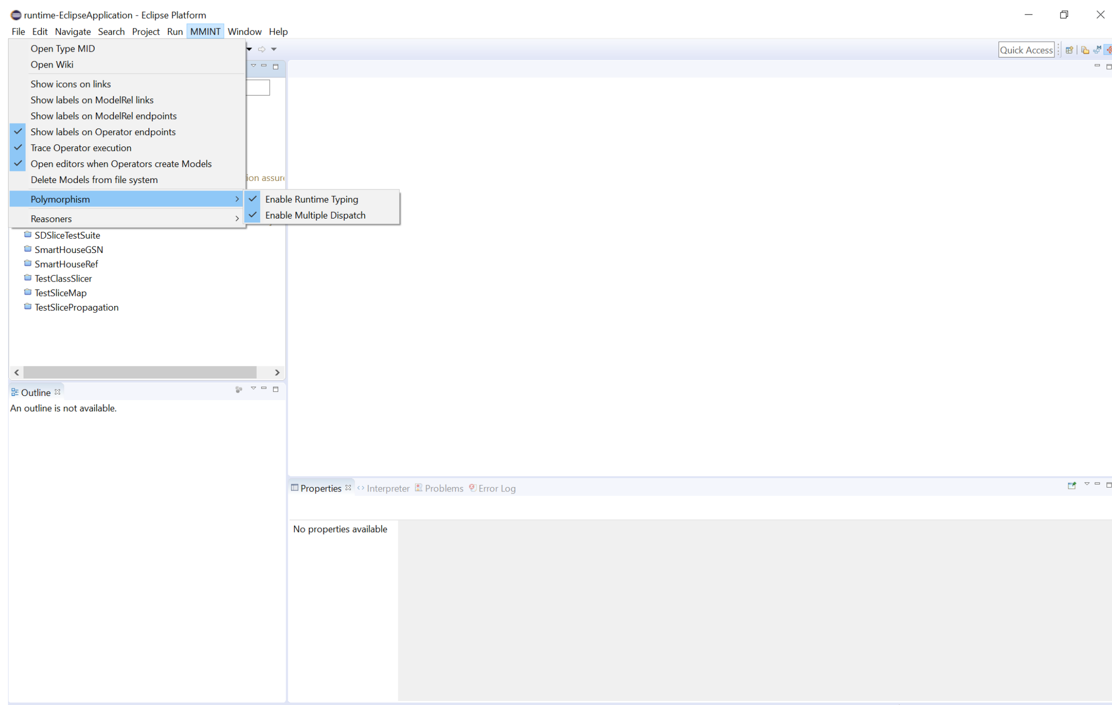


Fig. 2.2 A screenshot showing the menu for configuring MMINT.

Inspecting the Type MID

The Type MID is a mega-metamodel showing all the model types, relation types and operators that have been incorporated into and can be used in MMINT. To inspect it, select MMINT > Open Type MID from the menu bar (see Fig. 2.3).

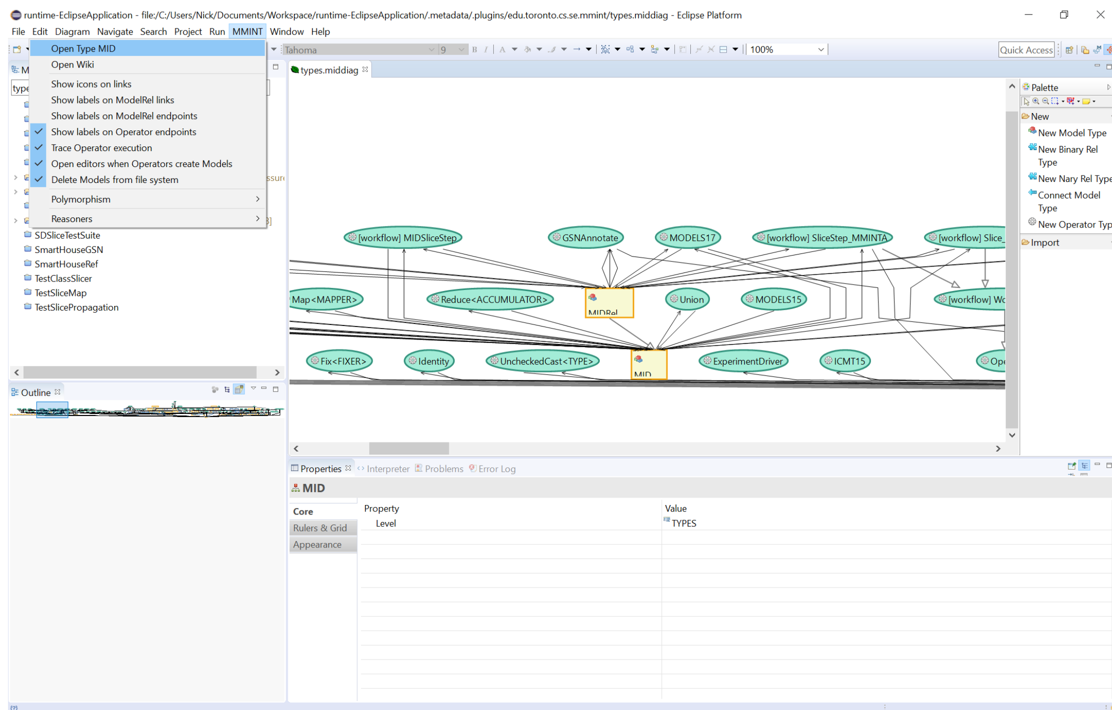


Fig. 2.3 A screenshot showing the Type MID and the menu item for opening it.

Creating a New Project

1. Select File > New > Project > Sirius > Modeling Project (see Fig. 2.4).

2. Give the project a name (e.g. HelloWorld) and choose a location for the project if desired.

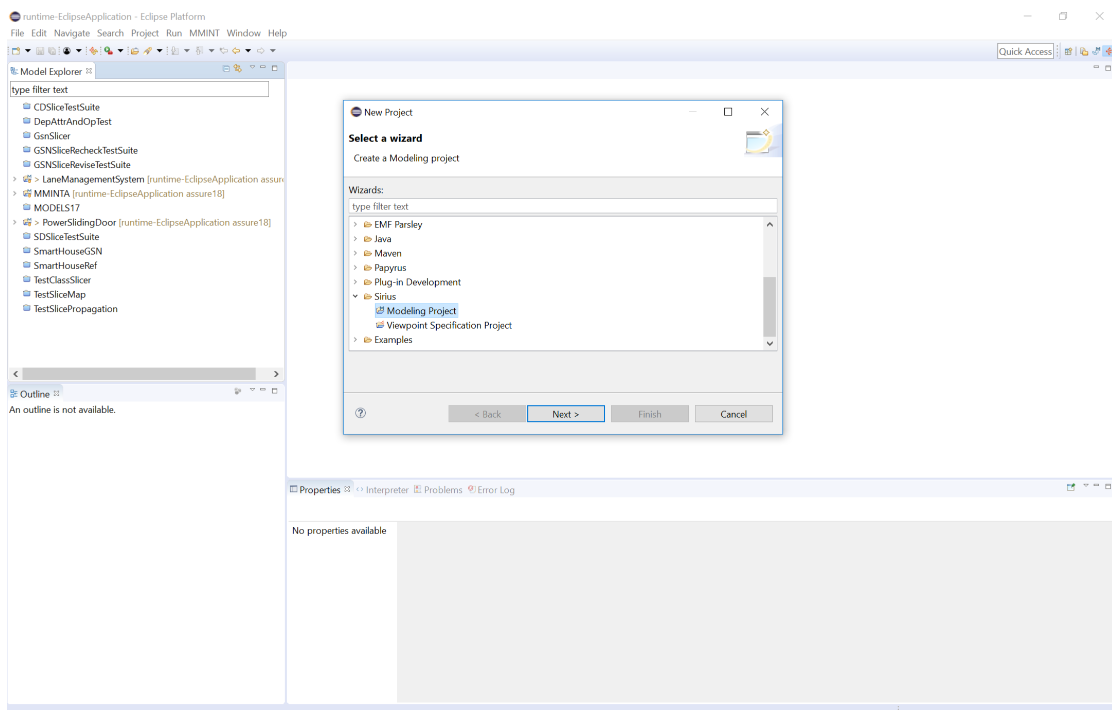


Fig. 2.4 A screenshot showing the type of projects to create for MMINT.

Creating a Megamodel

The following instructions describe how to create megamodels in MMINT. Note that megamodels are also referred to as **instance MID**s (Model Interconnection Diagrams) in MMINT, but to clearly distinguish between instance MIDs for megamodels and workflow MIDs for workflows, this manual will mainly use the terms "megamodel" (and "workflow") instead.

1. Right-click on a project (e.g. HelloWorld).
2. Select New > Other, and search for Instance MID (see Fig. 2.5).
3. Choose the appropriate parent directory and file name for the MID diagram (e.g. HelloMegamodel.middiag).[A] Click "Next" to continue.
4. Choose the appropriate parent directory and file name for the MID domain model underlying the MID diagram (e.g. HelloMegamodel.mid).[B] Click "Finish" to create new, empty megamodel.
5. Use the MID Editor to populate the megamodel.

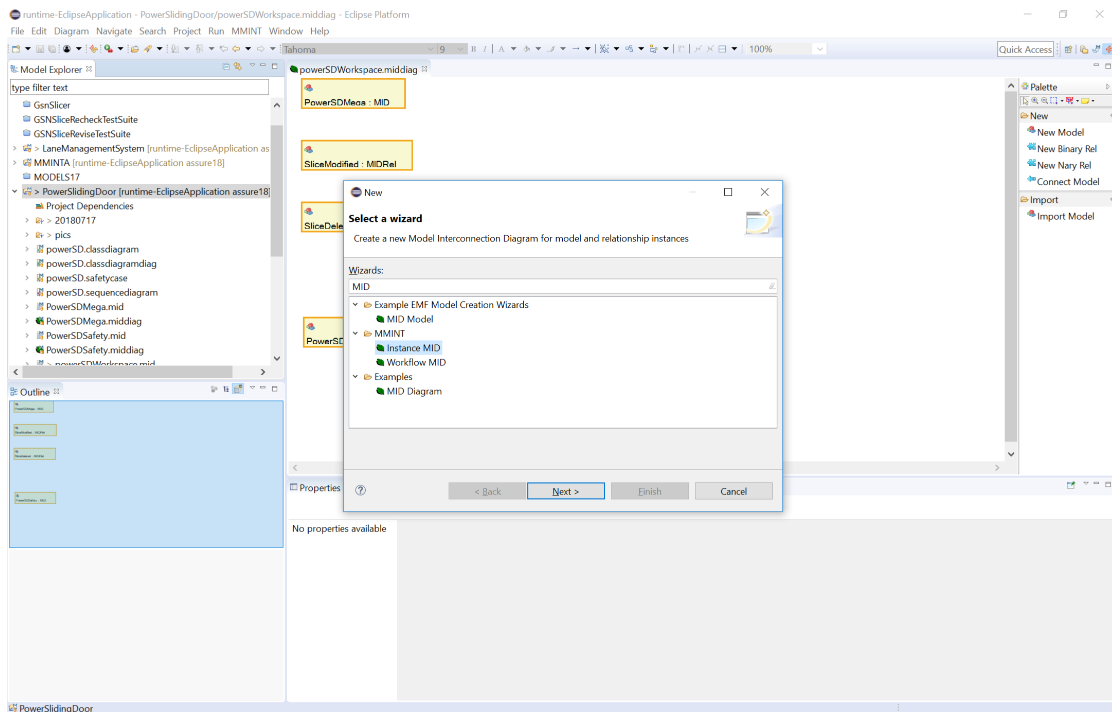


Fig. 2.5 A screenshot showing the creation of an instance MID (i.e. a megamodel).

Using the MID Editor

To create a new model using the MID editor:

1. Select "New Model" on the palette and click on an empty space in the MID.
2. Search for the desired model and editor (e.g. ClassDiagram representation).
3. Follow the instructions specific to the selected model and editor. E.g. for Sirius:
    1. Give the model an appropriate name (e.g. HelloClassDiagram).
    2. Choose the root of the meta-model (e.g. Class Diagram) as the model object.

To import an existing model into a MID:

1. Select "Import Model" on the palette and click on an empty space in the MID.
2. Search for and select the desired model to import.

To create a reference to a model in a different MID:

1. Right-click the MID and select "Create Shortcut…"
2. Search for and select the model to reference to.

To create a new binary model relation inside a MID:

1. Select "New Binary Rel" on the palette.

2. Click and hold on the source model of the relation.

3. Drag a connection to the target model of the relation.

4. Use the ModelRel editor to edit which specific model elements are related.

To create a new n-ary model relation inside a MID:

1. Select "New Nary Rel" on the palette and click on the MID.

2. Choose "ModelRel" as the model relationship type, and give it a name.

3. Select "Connect Model".

4. Click on the model relation and drag to (one of) the model or model references to be related.

5. Repeat until all appropriate models and references are connected by the model relation.

6. Use the ModelRel editor to edit which specific model elements are related.

To operate on a collection of models inside a MID:

1. Select all models and relations required as inputs to the operator (using Ctrl-Click). [C]

2. Right-click and select the desired operator under MMINT > Run Operator (see Fig. 2.6). For higher-order operators, further operators must be selected as input.

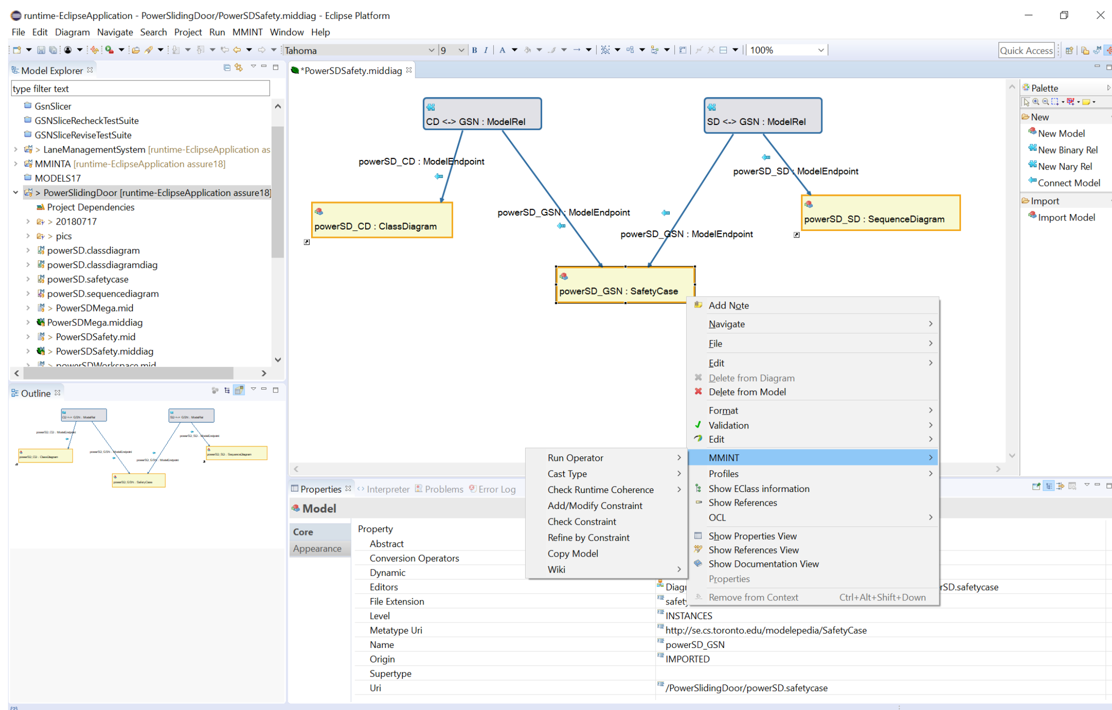


Fig. 2.6 A screenshot showing a megamodel and a list of actions that can be performed on the selected "powerSD_GSN" safety case model.

Using the ModelRel Editor

The ModelRel editor shows which model elements are related to each other in a model relation. In a newly created model relation, no elements are related to each other, thus the editor would display a collection of empty yellow boxes, one for each model connected by the relation. The following instructions describe how these boxes can be populated with model elements that can then be related together (see Fig. 2.7).

1. Double-click on a model relation to open the editor.
2. Open the "Outline" view in Eclipse if not open already.
    1. Select on the menu bar Window > Show View > Other...
    2. Look for and select the Outline view.
3. Add model elements to the relation.
    1. Find the desired model element in the Outline view.
    2. Click and drag the desired model element from the Outline view onto the appropriate yellow box.
4. Connect the model elements together as appropriate.
    - For a binary mapping between model elements:
        1. Select "New Binary Mapping" in the palette.
        2. Drag from one model element to the other. [D]
    - For an n-ary mapping:
        1. Select "New Nary Mapping" in the palette and click on an empty space.
        2. Give the mapping an optional name.
        3. Select "Connect Model Element" in the palette.
        4. Drag from the mapping to the appropriate model element.
        5. Repeat the previous two steps as appropriate.

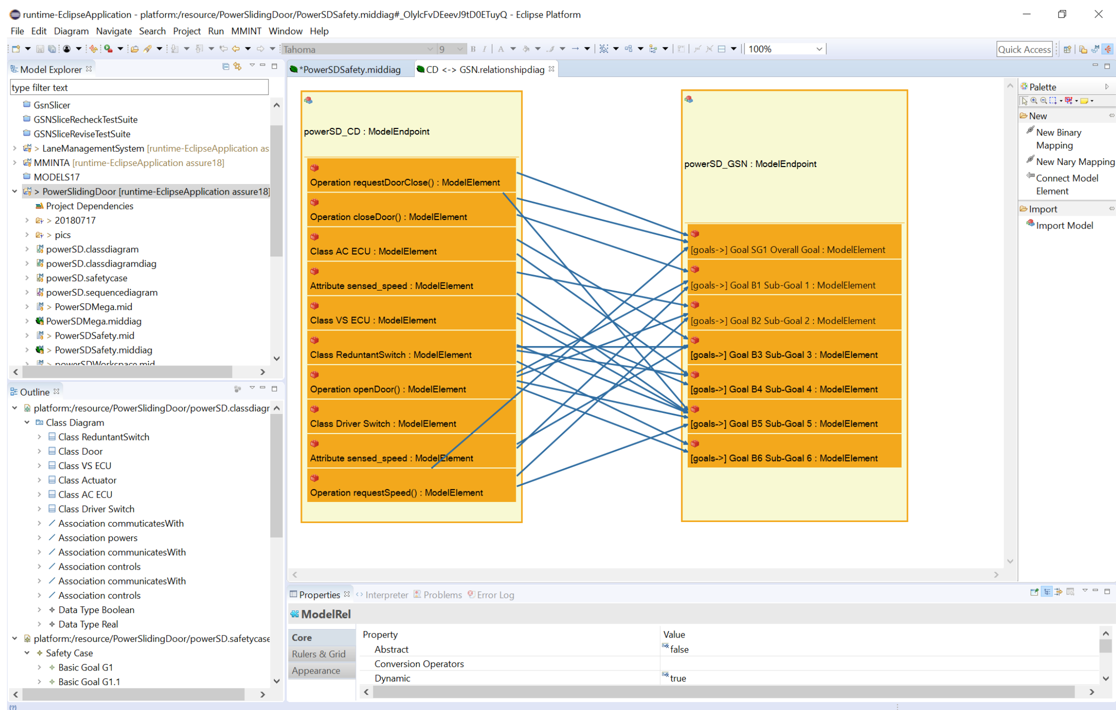


Fig. 2.7 A screenshot showing the ModelRel editor. The bottom left shows the Outline view which can be used to populate the yellow boxes with model elements. In this case, all model elements are related to each other by binary mappings.

Creating a Workflow

1. Right-click on a project (e.g. HelloWorld).
2. Select New > Other, and search for the Workflow MID (see Fig. 2.8).
3. Choose the appropriate parent directory and file name for the MID diagram (e.g. sayHello.middiag). Click "Next" to continue.
4. Choose the appropriate parent directory and file name for the MID domain model underlying the MID diagram (e.g. sayHello.mid). Click "Finish" to create new, empty workflow.
5. Use the Workflow Editor to populate the workflow.

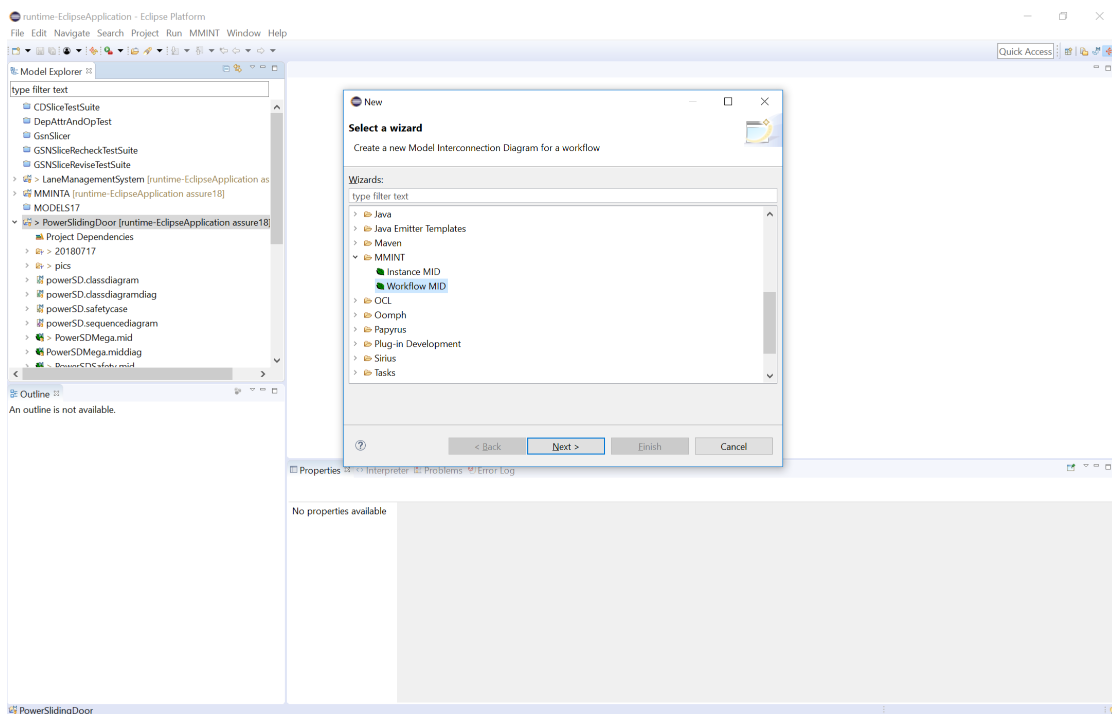


Fig. 2.8 A screenshot showing the wizard for creating a workflow MID (as opposed to an instance MID).

Using the Workflow Editor

Workflows are visualised in the same manner as megamodels, but the models in workflows represent the types of input, output and intermediary models that are instantiated when the workflow is executed. To create a new workflow:

1. Add a new input model to the workflow.
    1. Click on "New Model" on the palette and then on an empty area of the workflow.
    2. Select the desired model type.
2. Repeat the first step until all input models are added. The order the input models are added to the workflow determines the order in which the models must be selected to be operated on.
3. Follow the steps of operating on megamodels (see Using the MID Editor) to add the appropriate operators to the workflow (see Fig. 2.9). Repeating this process creates a series of operator applications starting with the input models and ending with the output models.

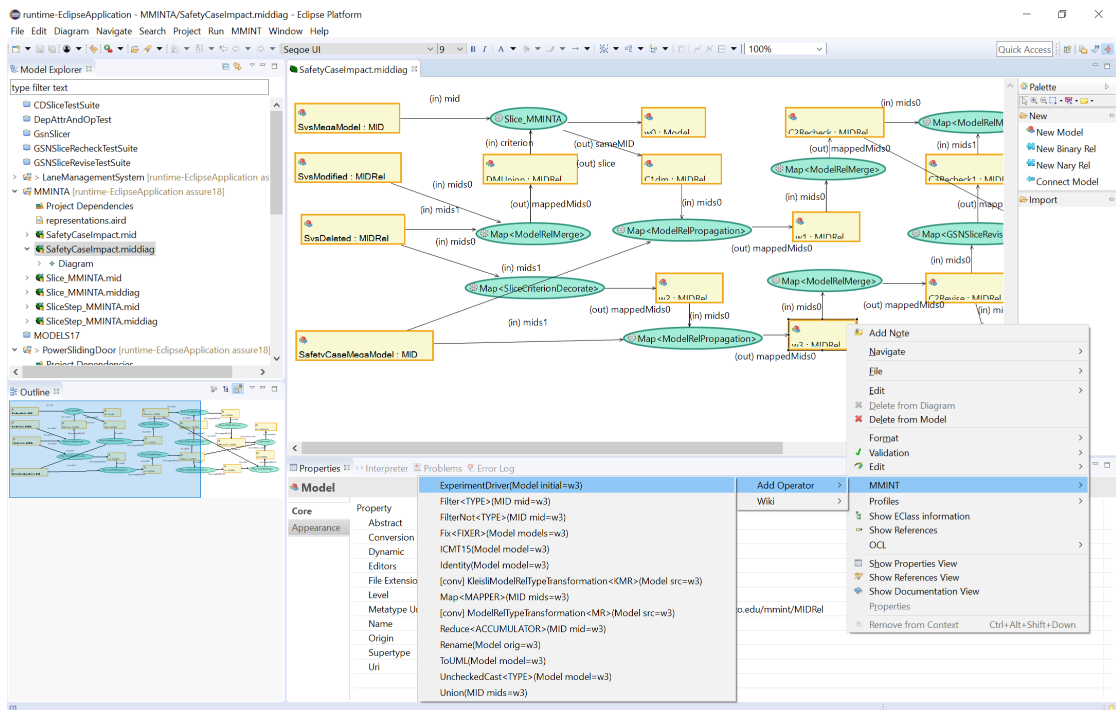


Fig. 2.9 A screenshot showing the workflow editor. In this case, the workflow accepts two MIDs and two MIDRels as inputs, and a new model operator is being added to w3.

Registering New Workflows

Before they can be used like other operators, newly created workflows must be registered in MMINT by following the instructions below:

1. Open the Type MID.

2. Click on "New Operator Type" in the palette and select an empty spot in the Type MID.

3. Look for and select the workflow to add to the MID (see Fig. 2.10).

4. Change the name of the workflow if desired.

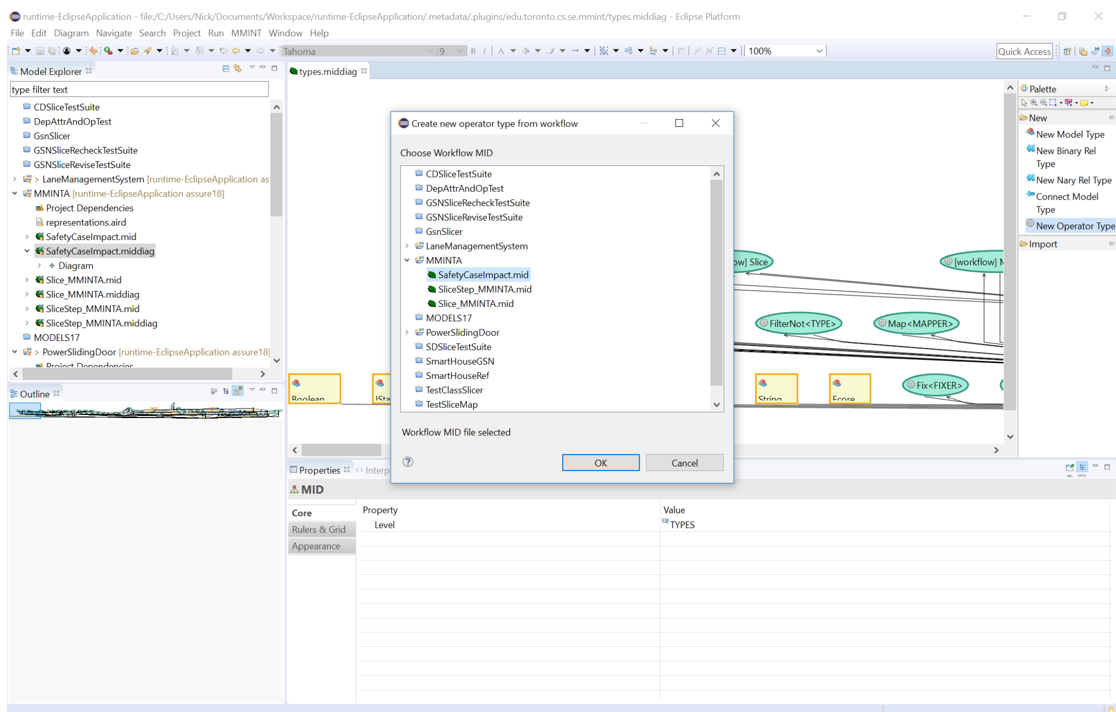


Fig. 2.10 A screenshot showing the window for adding new workflow operators to the Type MID.

A: The MID diagram is the graphical visualisation of a megamodel, which is built on top of the Graphical Modeling Framework (GMF) (http://www.eclipse.org/modeling/gmp) in Eclipse.

B: The MID domain model is the actual instance of a megamodel, which is built on top of the Eclipse Modeling Framework (EMF) (http://www.eclipse.org/modeling/emf).

C: The order in which the models and relations are selected must be the same as the order specified for the operator.

D: Binary mappings are directional in MMINT.

3 Using MMINT-A

Setting Up MMINT-A

MMINT-A comprises a set of extensions to MMINT, viz.:

- A safety case metamodel (and accompanying editor)
- Safety case slicers for propagating change impact
- A workflow for performing change impact assessment on the system models

- The complete safety case change impact assessment workflow

The metamodel and slicers are automatically incorporated into MMINT, and thus do not require further action from the user. However, the workflows must still be incorporated into MMINT, the instructions for which are as follows:

1. Go to https://github.com/nlsfung/MMINT_Examples/tree/assure18 (https://github.com/nlsfung/MMINT_Examples/tree/assure18)
2. Download the MMINTA directory into MMINT's runtime Eclipse directory.
3. Start up MMINT (https://github.com/nlsfung/MMINT/wiki/02-Using-MMINT#starting-up).
4. Import the MMINTA project.
5. Add to the Type MID (https://github.com/nlsfung/MMINT/wiki/02-Using-MMINT#registering-new-workflows) the workflows (see Fig. 3.1):
    - SliceStep_MMINTA.mid
    - Slice_MMINTA.mid
    - SafetyCaseImpact.mid.

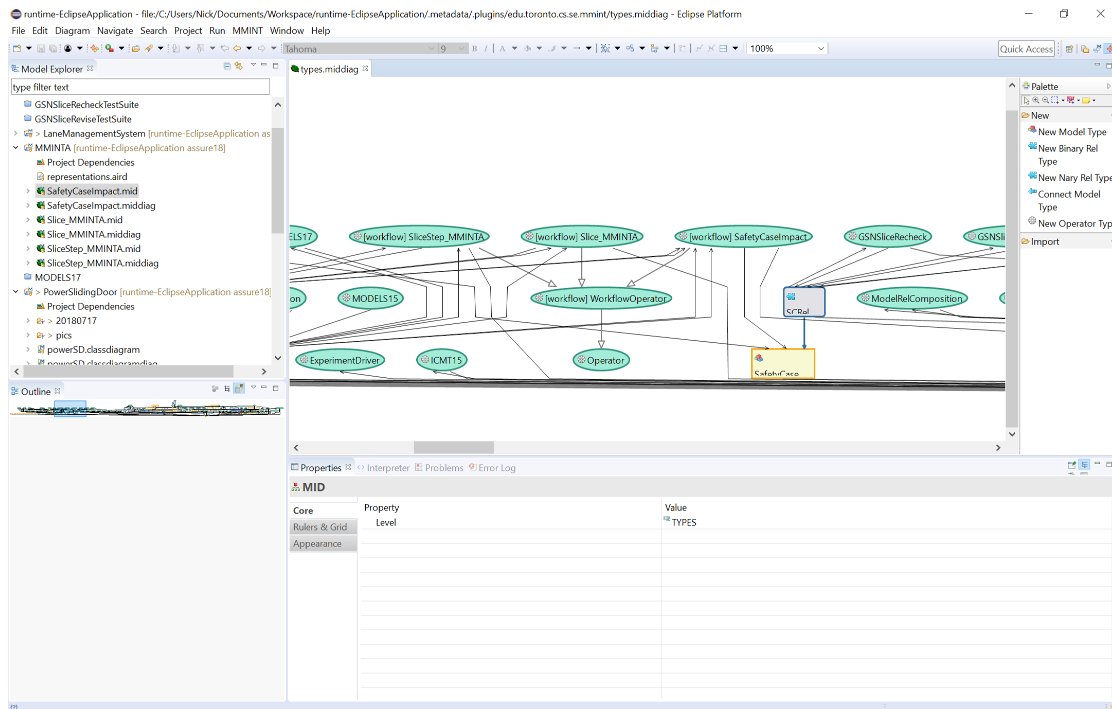


Fig. 3.1 A screenshot showing the Type MID with the workflows for MMINT-A added.

Preparing the Assessment

To apply the safety case change impact assessment workflow (SafetyCaseImpact.mid), the following inputs are

required (see Fig. 3.2):

- The system models, which can include any of the following:
  - Class Diagram
  - Sequence Diagram
  - State Machine
- The system safety case
- The system megamodel, which comprises:
  - The system models
  - 2-ary relations between pairs of models expressing the dependencies between them [A]
- The modified elements megamodel, which identify the system elements to be modified and is created as follows using the MID editor (https://github.com/nlsfung/MMINT/wiki/02-Using-MMINT#using-the-mid-editor):
  1. Create references to the system models to be modified.
  2. Add a unary relation to each model.
  3. Use the ModelRel editor (https://github.com/nlsfung/MMINT/wiki/02-Using-MMINT#using-the-modelrel-editor) to populate each unary relation with the modified model elements.
- The deleted elements megamodel, which identify the system elements to be deleted and is created as follows using the MID editor (https://github.com/nlsfung/MMINT/wiki/02-Using-MMINT#using-the-mid-editor):
  1. Create references to the system models to be modified.
  2. Add a unary relation to each model.
  3. Use the ModelRel editor (https://github.com/nlsfung/MMINT/wiki/02-Using-MMINT#using-the-modelrel-editor) to populate each unary relation with the deleted model elements.
- The megamodel for the safety case containing:
  - The safety case
  - A reference to each system model
  - A 2-ary relation connecting each system model with the safety case
- The megamodel containing the four megamodels above

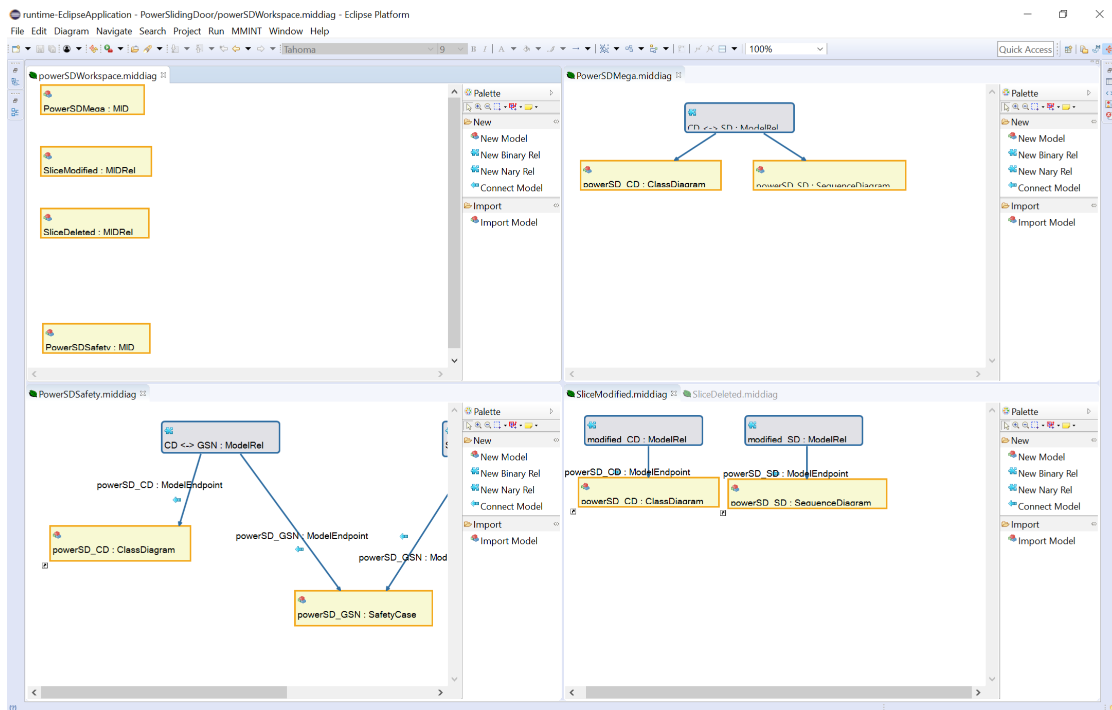


Fig. 3.2 A screenshot showing some of the megamodels required for performing safety case change impact assessment using MMINT-A. From left to right, top to bottom, the megamodels shown are: the overall megamodel, the system megamodel, the modified elements megamodel and the safety case megamodel.

Performing the Assessment

1. Open the megamodel containing the four megamodels.
2. Use Ctrl-Click to select, in order:
   - The system megamodel
   - The megamodel of system elements to be modified
   - The megamodel of system elements to be deleted
   - The megamodel for the safety case
3. Right-click the selection.
4. Select MMINT > Run Operator > [workflow] SafetyCaseImpact (see top of Fig. 3.3).
5. Double-click the output to open the annotated safety case (see bottom of Fig. 3.3).

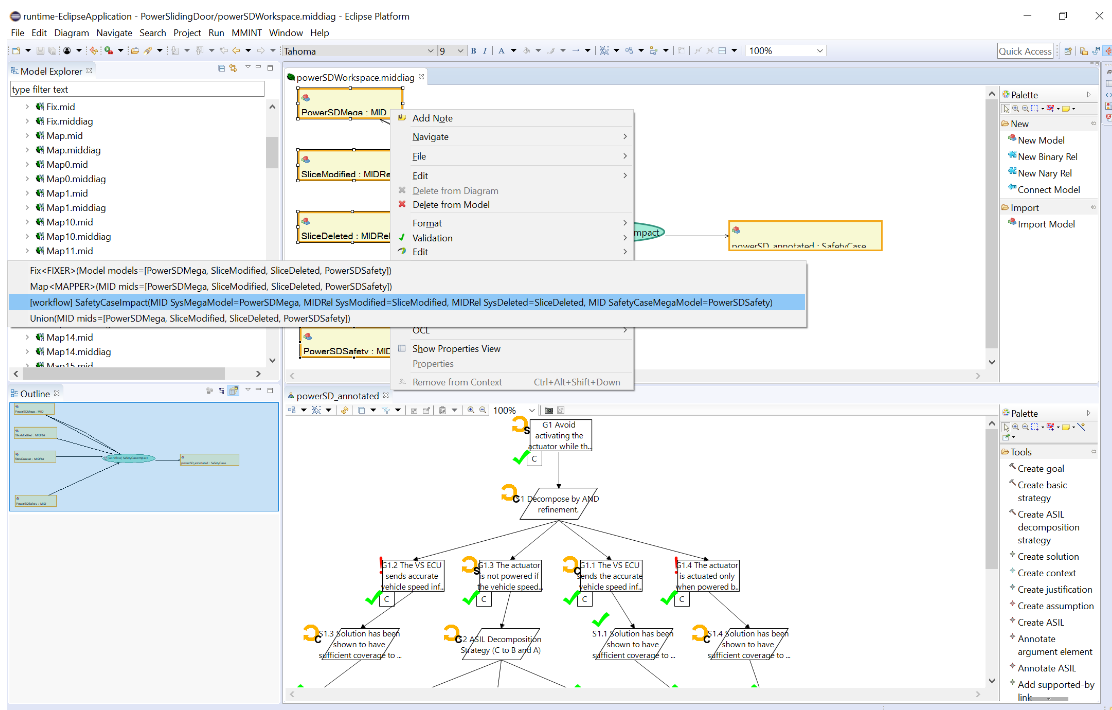


Fig. 3.3 A screenshot showing, at the top, the safety case change impact assessment workflow being selected for execution and, at the bottom, the resulting annotated safety case.

Analysing the Results

Apart from the standard graphical view of safety cases, MMINT-A can also generate tables summarising (see Fig. 3.4):

- The amount of impact the given change has on the safety case.
- The source of each annotation on the safety case.

The following instructions describe how these tables can be created:

1. Go to the Model Explorer in Eclipse, and expand the annotated safety case.
2. Right-click on "Safety Case".
3. Select the appropriate representation to create in "New Representation".
   - "new SafetyCaseRepresentation" creates another graphical view of the safety case.
   - "new SafetyCaseStatistics" creates a table summarising the number of different annotations.
   - "new SafetyCaseImpactTrace" creates a table listing the annotations and their source.

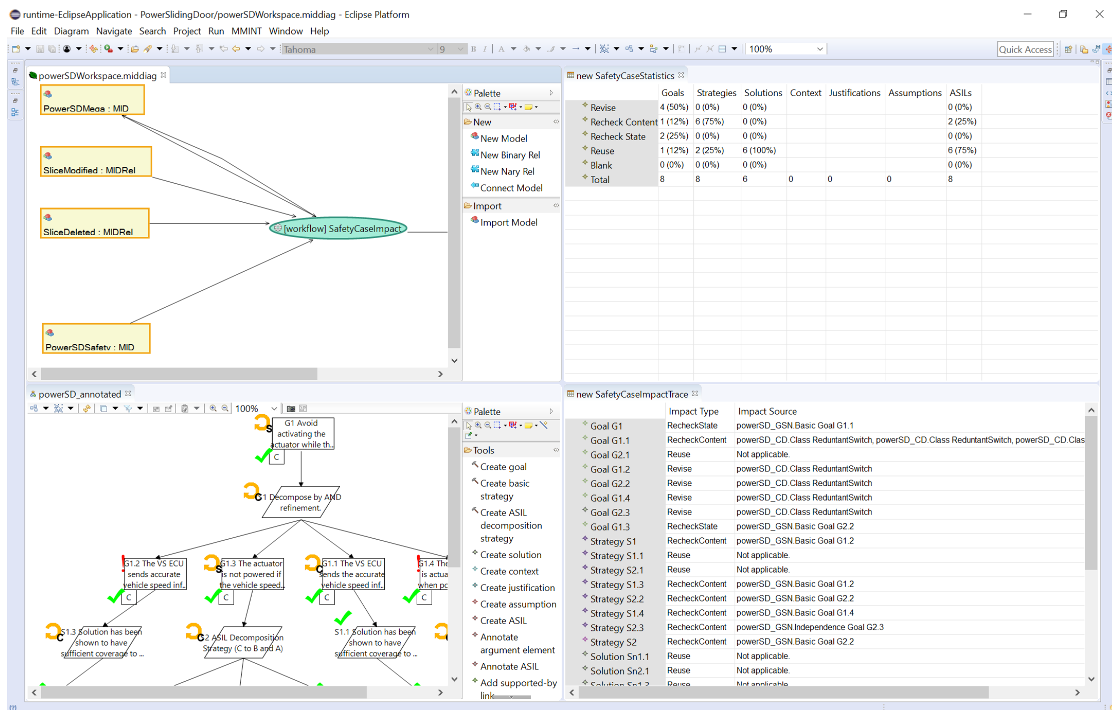


Fig. 3.4 A screenshot showing, from top to bottom, left to right, the overall megamodel for performing safety case change impact assessment, a table summarising the number of different annotations, the annotated safety case, and a table summarising the source of each annotation.

Sample Projects

https://github.com/nlsfung/MMINT_Examples/tree/assure18 (https://github.com/nlsfung/MMINT_Examples /tree/assure18) contains some sample projects for executing the safety case change impact assessment workflow. These projects, and their main models and megamodels, are as follows:

- PowerSlidingDoor (PSD)
  - **powerSD.classdiagram** The class diagram for the PSD system
  - **powerSD.safetycase** The safety case for the PSD system
  - **powerSD.sequencediagram** The sequence diagram for the PSD system
  - **PowerSDMega.mid(diag)** The PSD system megamodel
  - **PowerSDSafety.mid(diag)** The PSD safety case megamodel
  - **powerSDWorkspace.mid(diag)** The megamodel for running the workflow
  - **SliceDeleted.mid(diag)** The megamodel of deletions
  - **SliceModified.mid(diag)** The megamodel of modifications

- LaneManagementSystem (LMS)
    - **lms_cd.classdiagram** The LMS class diagram
    - **lms_cia.mid(diag)** The megamodel for running the workflow
    - **lms_deleted.mid(diag)** The megamodel of deletions
    - **lms_mega.mid(diag)** The LMS megamodel
    - **lms_modified.mid(diag)** The megamodel of modifications
    - **lms_safety.mid(diag)** The LMS safety case megamodel
    - **lms_sc.safetycase** The LMS safety case
    - ***.sequencediagram** The sequence diagrams for four different LMS sub-systems
    - ***.statemachine** The state machines for four different LMS sub-systems

A: MMINT-A only supports binary relations that are created as n-ary relations with two connections.

4 Extending MMINT

Adding a New Metamodel (and Basic Editor)

The following instructions describe how new metamodels (and their default editors) in Eclipse can be plugged into MMINT. These metamodels are created using EMF (https://www.eclipse.org/modeling/emf/) [A] and comprise three projects, e.g.:

- edu.toronto.cs.se.modelepedia.classdiagram
- edu.toronto.cs.se.modelepedia.classdiagram.edit
- edu.toronto.cs.se.modelepedia.classdiagram.editor

The first project is the main one containing the Ecore specification of the model as well as the generated code. The other two projects (*.edit and *.editor) create an Eclipse plug-in for the metamodel, allowing it to be instantiated.

The *.edit project need not be modified for MMINT, but to register the metamodel (the main project) with MMINT (see Fig. 4.1):

1. Open the plugin.xml file for the project.
2. Select the "Dependencies" tab
3. Add "edu.toronto.cs.se.mmint" as a dependency (required plug-in)
4. (Optional) Select the plugin and remove the version number from its properties

5. Select the "Extensions" tab

6. Add "edu.toronto.cs.se.mmint.models" as an extension

7. Select the added extension and expand it two levels down. (e.g. edu.toronto.cs.se.mmint.models > (modelType) > edu.toronto.cs.se.modelepedia.classdiagram.type1)

8. Select the result (edu.toronto.cs.se.modelepedia.classdiagram.type1) to inspect its extension element details.

9. Change the name of the model type as appropriate (e.g. ClassDiagram).

10. Set the URI to be the same as that of the metamodel, which can be found by:

    1. Expand the extension "org.eclipse.emf.ecore.generated_package"

    2. Select the result (e.g. ClassDiagramPackage).

    3. Inspect its extension element details for its URI. (e.g. http://se.cs.toronto.edu/modelepedia/ClassDiagram (http://se.cs.toronto.edu/modelepedia/ClassDiagram))

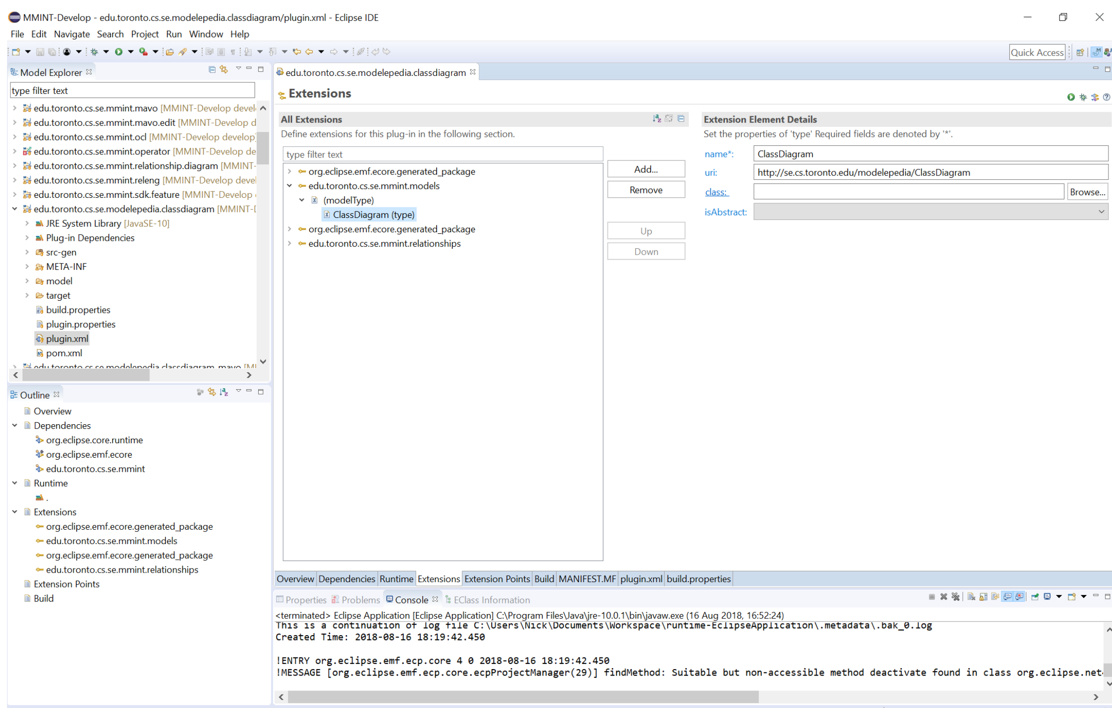


Fig. 4.1 A screenshot showing the extension element details for registering a class diagram metamodel with MMINT.

To register the *.editor project with MMINT:

1. Open the plugin.xml file for the project.

2. Select the "Dependencies" tab

3. Add "edu.toronto.cs.se.mmint" as a dependency (required plug-in)

4. (Optional) Select the plugin and remove the version number from its properties
5. Select the "Extensions" tab
6. Add "edu.toronto.cs.se.mmint.editors" as an extension
7. Select the added extension and expand it one level down. (e.g. edu.toronto.cs.se.mmint.editors > edu.toronto.cs.se.modelepedia.classdiagram.editor.editorType1 (editorType))
8. Select the result (edu.toronto.cs.se.modelepedia.classdiagram.editor.editorType1) to inspect the extension element details of the editor type.
9. Set the modelTypeUri to be the same as the metamodel URI. (e.g. http://se.cs.toronto.edu/modelepedia /ClassDiagram (http://se.cs.toronto.edu/modelepedia/ClassDiagram))
10. Set the id to be the same as the editor ID, which can be found by:
    1. Expand the extension "org.eclipse.ui.editors".
    2. Select the result (e.g. ClassDiagram Model Editor)
    3. Inspect its extension element details for its id (e.g. edu.toronto.cs.se.modelepedia.classdiagram.presentation.ClassDiagramEditorID)
11. Set the wizardId to be the same as the wizard ID of the editor, which can be found by:
    1. Expand the extension "org.eclipse.ui.newWizards".
    2. Select the second result (e.g. ClassDiagram Model (wizard))
    3. Inspect its extension element details for its id (e.g. edu.toronto.cs.se.modelepedia.classdiagram.presentation.ClassDiagramModelWizardID)
12. Expand the added extension ("edu.toronto.cs.se.mmint.editors") one further level down. (e.g. to edu.toronto.cs.se.modelepedia.classdiagram.editor.type1)
13. Select the result to inpect the extension element details of the editor.
14. Give the editor an arbitrary name (e.g. ClassDiagram tree)
15. Give the editor a unique URI (e.g. http://se.cs.toronto.edu/modelepedia/Tree_ClassDiagram (http://se.cs.toronto.edu/modelepedia/Tree_ClassDiagram))

Adding a New Sirius Editor

The following instructions describe how new editors created using Sirius (https://www.eclipse.org/sirius/) [B] can be plugged into MMINT. Each of these editors typically comprise a single project, e.g.:

- edu.toronto.cs.se.modelepedia.classdiagram.design

To register the .design project with MMINT:

1. Open the plugin.xml file for the project.

2. Select the "Dependencies" tab
3. Add "edu.toronto.cs.se.mmint" as a dependency (required plug-in)
4. (Optional) Select the plugin and remove the version number from its properties
5. Select the "Extensions" tab
6. Add "edu.toronto.cs.se.mmint.editors" as an extension
7. Select the added extension and expand it one level down. (e.g. edu.toronto.cs.se.mmint.editors > edu.toronto.cs.se.modelepedia.classdiagram.editor.editorType1 (editorType))
8. Select the result (edu.toronto.cs.se.modelepedia.classdiagram.editor.editorType1) to inspect the extension element details of the editor type (see top of Fig. 4.2).
9. Set the modelTypeUri to be the same as the metamodel URI. (e.g. http://se.cs.toronto.edu/modelepedia /ClassDiagram (http://se.cs.toronto.edu/modelepedia/ClassDiagram))
10. Give the editor type an arbitrary ID (e.g. edu.toronto.cs.se.modelepedia.classdiagram.design)
11. Set isDiagram to true.
12. Set the wizardId to be the same as the wizard ID of the default editor, which can be found by:
    1. Expand the extension "org.eclipse.ui.newWizards".
    2. Select the second result (e.g. ClassDiagram Model (wizard))
    3. Inspect its extension element details for its id (e.g. edu.toronto.cs.se.modelepedia.classdiagram.presentation.ClassDiagramModelWizardID)
13. Expand the added extension ("edu.toronto.cs.se.mmint.editors") one further level down. (e.g. to edu.toronto.cs.se.modelepedia.classdiagram.editor.type1)
14. Select the result to inpect the extension element details of the editor (see bottom of Fig. 4.2).
15. Give the editor an arbitrary name (e.g. ClassDiagram representation)
16. Give the editor a unique URI (e.g. http://se.cs.toronto.edu/modelepedia/Representation_ClassDiagram (http://se.cs.toronto.edu/modelepedia/Representation_ClassDiagram))

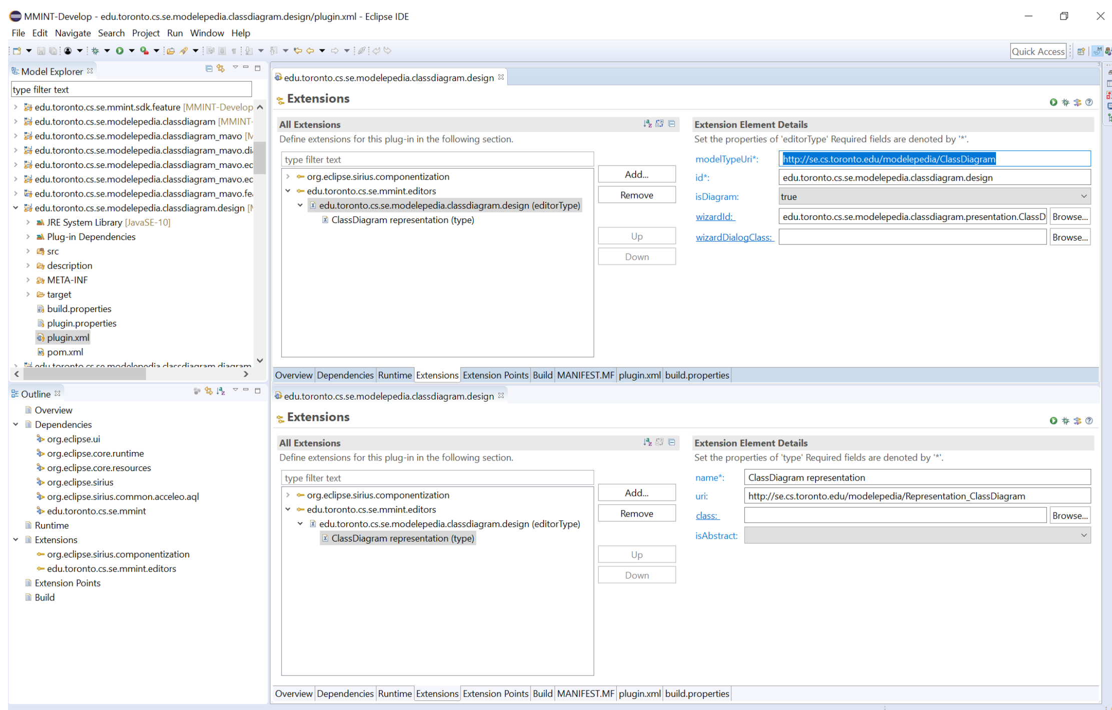


Fig. 4.2 A screenshot showing the extension element details for registering a Sirius editor (for class diagrams) with MMINT.

Adding a New Operator

The following instructions describe how to register a new model operator with MMINT.[C]

1. Open the plugin.xml file for the operator's project (e.g. edu.toronto.cs.se.modelepedia.classdiagram.operator).
2. Select the "Dependencies" tab
3. Make sure "edu.toronto.cs.se.mmint" is added as a dependency (required plug-in)
4. Make sure "edu.toronto.cs.se.mmint.operator" is added as a dependency (required plug-in)
5. Select the "Extensions" tab
6. Add "edu.toronto.cs.se.mmint.operators" as an extension
7. Select the added extension and expand it two levels down. (e.g. edu.toronto.cs.se.mmint.operators > (operatorType) > edu.toronto.cs.se.modelepedia.classdiagram.operator.type1 (type))
8. Select the result to inspect the extension element details of the editor type (see Fig. 4.3).
9. Give the operator type an arbitrary name. (e.g. Slice)
10. Give the operator type a unique uri. (e.g. http://se.cs.toronto.edu/modelepedia/Operator_CDSlice (http://se.cs.toronto.edu/modelepedia/Operator_CDSlice))

11. Search for and specify the class implementing the operator (e.g. edu.toronto.cs.se.modelepedia.classdiagram.operator.CDSlice)

12. Specify the supertype of the operator (if any) by:
    1. Right-click the extension element (edu.toronto.cs.se.modelepedia.classdiagram.operator.type1)
    2. Select New > superType.
    3. Set its URI to be the same as the URI of the supertype's URI (e.g. http://se.cs.toronto.edu/mmint /Operator_Slice (http://se.cs.toronto.edu/mmint/Operator_Slice))

13. Right-click on "(operatorType)" and select New > inputs.

14. Register each input to the operator (in order) by:
    1. Right-click on "(inputs)" and select New > parameter
    2. Expand "(parameter)" one level down.
    3. Inspect the extension element details of the parameter type (e.g. edu.toronto.cs.se.modelepedia.classdiagram.operator.type1)
    4. Give the parameter type an arbitrary name (e.g. criterion)
    5. Give the parameter type a unique URI (e.g. http://se.cs.toronto.edu/modelepedia/Operator_CDSlice-CDRel/criterion (http://se.cs.toronto.edu/modelepedia/Operator_CDSlice-CDRel/criterion))
    6. Specify the supertype of the parameter type (if any) by following analogous instructions as before.
    7. Inspect the extension element details of the parameter type end point (e.g. edu.toronto.cs.se.modelepedia.classdiagram.operator.typeEndpoint1)
    8. Set its targetTypeUri to be the URI of the target type of the parameter (e.g. http://se.cs.toronto.edu /modelepedia/CDRel (http://se.cs.toronto.edu/modelepedia/CDRel))

15. Register each output from the operator (in order) by following analogous instructions for the previous step.

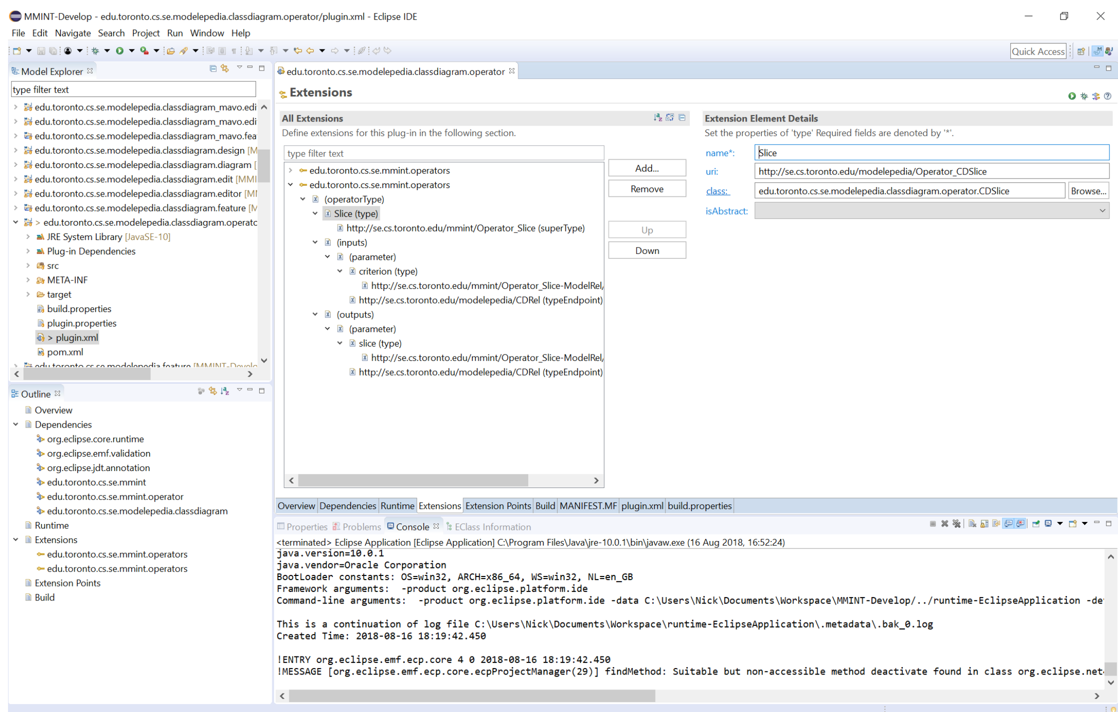


Fig. 4.3 A screenshot showing the addition of a "Slice" operation in MMINT.

A: A tutorial on using EMF can be found on vogella.com (http://www.vogella.com/tutorials/EclipseEMF/article.html). However, note that it is not up-to-date as the visual editor for creating EMF models are now based on Sirius instead of GMF.

B: A tutorial on using Sirius can be found on eclipse.org (http://www.eclipse.org/sirius/getstarted.html).

C: The details of the MMINT API for implementing model operators are beyond the scope of this manual.

.