

University of Southampton Research Repository ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

UNIVERSITY OF SOUTHAMPTON

Safety Cases for the Formal Verification of Automatically Generated Code

by

Nurlida Basir

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

July 2010

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Nurlida Basir

Model-based development and automated code generation are increasingly used for actual production code, in particular in mathematical and engineering domains. However, since code generators are typically not qualified, there is no guarantee that their output is correct or even safe. Formal methods which are based on mathematically-based techniques have been proposed as a means to improve software quality by providing formal safety proofs as explicit evidence for the assurance claims. However, the proofs are often complex and may also be based on assumptions and reasoning principles that are not justified. This causes concerns about the trustworthiness of the proofs and hence the assurance claims on the safety of the program. This thesis presents an approach to systematically and automatically construct comprehensive safety cases using the Goal Structuring Notation from a formal analysis of automatically generated code, based on automated theorem proving, and driven by a set of safety requirements and properties. We also present an approach to systematically derive safety cases that argue along the hierarchical structure of systems in model-based development. This core safety case is extended by separately specified auxiliary information from other verification and validation activities such as testing. The thesis also presents an approach to develop safety cases that correspond to the formal proofs found by automated theorem provers and that reveal the underlying proof argumentation structure and top-level assumptions. The resulting safety cases will make explicit the formal and informal reasoning principles, and reveal the top-level assumptions and external dependencies that must be taken into account in demonstrating software safety. The safety cases can be thought as “structured reading guide” for the software and the safety proofs that provide traceable arguments on the assurance provided. The approach has been illustrated on code generated using Real-Time Workshop for Guidance, Navigation, and Control (GN&C) systems of NASA’s Project Constellation and on code for deep space attitude estimation generated by the AutoFilter system developed at NASA Ames.

Contents

Glossary	xiii
Declaration of Authorship	xvii
List of Publications	xix
Acknowledgements	xxi
1 Introduction	1
1.1 Introduction	1
1.2 Problem Description	2
1.3 Outline of the Solution	3
1.4 Thesis Proposition	6
1.5 Thesis Structure	6
2 Background	9
2.1 Automated Code Generation	9
2.2 Generator Assurance	13
2.3 Formal Methods and Formal Logic	15
2.3.1 Formal Methods	15
2.3.2 Formal Logic	16
2.3.3 Formal Proofs	20
2.4 Hoare-Style Verification Systems	21
2.4.1 Proof-Carrying Code	21
2.4.2 Formal Software Safety Certification	23
2.4.3 Proof-Carrying Code vs Formal Software Safety Certification	26
2.5 Hazard Analysis	27
2.5.1 Hazard Analysis Terminology	27
2.5.2 Hazard Severity Classification Scheme	28
2.5.3 Hazard Analysis Models and Techniques	30
2.6 Argumentation	31
2.6.1 Explanation and Argumentation	31
2.6.2 Safety Cases	33
2.6.3 Goal Structuring Notation	35
2.6.4 Safety Case Patterns	38
2.6.5 Safety Case Assurance	40
2.7 Conclusions	41

3	Hazard Analysis for Formal Program Verification	43
3.1	Introduction	43
3.2	Hazard Analysis Methodology	44
3.2.1	Operational Description Process	45
3.2.2	Hazard Assessment Process	46
3.3	Hazard Severity Classification Scheme for Program Verification	47
3.4	Fault Tree Analysis for Formal Program Verification	49
3.5	Conclusions	52
4	Property-Oriented Safety Cases	55
4.1	Introduction	55
4.2	Constructing Safety Cases for Demonstrating a Safety Property	58
4.2.1	Tier I: Explaining the Safety Notion	58
4.2.2	Tier II: Safety of Program Variables	60
4.2.3	Tier III: Sufficiency of Safety Condition	61
4.3	Safety Case Patterns for Property-Oriented Safety Cases	62
4.3.1	Safety Case Pattern: Explaining the Safety Notion	63
4.3.2	Safety Case Pattern: Safety of Program Variables	63
4.3.3	Safety Case Pattern: Sufficiency of Safety Condition	64
4.4	Conclusions	64
5	Requirement-Oriented Safety Cases	77
5.1	Introduction	77
5.2	Constructing Safety Cases for Demonstrating a Safety Requirement	80
5.2.1	Arguing over Formalization of Safety Requirement	82
5.2.2	Sufficiency of Safety Conditions for Individual Variables	84
5.3	Safety Case Patterns for Requirement-Oriented Safety Cases	85
5.3.1	Safety Case Pattern: Formalization of Safety Requirement	85
5.3.2	Safety Case Pattern: Sufficiency of Safety Conditions for Individual Variables	86
5.4	Conclusions	86
6	Architecture-Oriented Safety Cases	93
6.1	Introduction	93
6.2	Constructing Safety Cases from the Formal Analysis of Hierarchical Systems	97
6.2.1	Arguing from System-Level Safety Requirements to Component-Level Safety Requirements	97
6.2.2	Arguing from Component-Level Safety Requirements to Source Code	100
6.2.3	Combining System-Level and Component-Level Safety Cases	101
6.3	Safety Case Patterns: Architecture-Oriented	102
6.3.1	Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements	102
6.3.2	Safety Case Pattern: Arguing from Component-Level Safety Requirements to Source Code	102
6.4	Conclusions	103
7	Proof-Oriented Safety Cases	111
7.1	Introduction	111

7.2	Converting Natural Deduction Proofs into Safety Cases	112
7.2.1	Conversion Process	112
7.2.1.1	Conjunctions	113
7.2.1.2	Disjunctions	114
7.2.1.3	Implications	115
7.2.1.4	Negations	115
7.2.1.5	Universal quantifiers	116
7.2.1.6	Existential quantifiers	117
7.2.1.7	Equalities	117
7.3	Hypothesis Handling	118
7.4	Application to Muscadet Prover	119
7.5	Proof Abstraction	122
7.6	Conclusions	125
8	Heterogenous Safety Cases	129
8.1	Introduction	129
8.2	Constructing Heterogenous Safety Cases	130
8.2.1	Tier I of Heterogenous Safety Case: Explaining the Safety Notion	130
8.2.2	Tier II of Heterogenous Safety Case: Arguing over Safety of Program	132
8.2.3	Tier III of Heterogenous Safety Case: Arguing over Sufficiency of Safety Condition	135
8.2.4	Tier IV of Heterogenous Safety Case: Arguing over Soundness of Formal Proof	136
8.3	Safety Case Evidence	137
8.4	Conclusions	139
9	Implementation and Preliminary Assessment of the Approach	141
9.1	Introduction	141
9.2	Automating the Safety Case Generation	142
9.2.1	Architecture of the Safety Case Generation	144
9.2.2	Implementation	145
9.3	Evaluation through Safety Case Checklist	151
9.4	Industrial Applications	153
9.5	Conclusions	156
10	Conclusions	157
10.1	Main Contributions	157
10.1.1	Fault Tree Analysis	158
10.1.2	Approach to Construct Safety Case for the Formal Program Verification of Auto-Generated Code	158
10.1.3	Software Safety Argument Patterns	160
10.1.4	Automatic Safety Case Generation Approach	161
10.2	Limitations	161
10.3	Future Work Directions	162
10.4	Concluding Remarks	163
A	Roles and Principles of FSSC Components	165

Bibliography

169

List of Figures

1.1	Approach: From Formal Proofs to Safety Cases	5
2.1	Main Tasks of a Code Generator [40]	10
2.2	Overview of Proof-Carrying Code [106]	22
2.3	Assignment Rules	23
2.4	Architecture of Formal Software Safety Certification System [44]	25
2.5	Principal Elements of Safety Case - Bishop and Bloomfield [29]	34
2.6	Principal Elements of Safety Case - Kelly [86]	34
2.7	The Steps of the GSN Construction Method [86]	37
2.8	The EUR RVSM Pre-Implementation Safety Case - Overall Argument (adapted from [56])	37
2.9	GSN Extensions to Represent Patterns and Their Relationships	38
2.10	Template for the Safety Pattern Document	39
3.1	Position of Error, Fault, Failure, Hazard and Accident in the Formal Program Verification (adapted from [146])	44
3.2	Hazard Analysis Methodology (adapted from [93])	45
3.3	Program Verification Hazard Decision Matrix	49
3.4	Fault Tree for Program Verification	53
4.1	Code Fragment and Generated Annotation [50]	57
4.2	Tier I of Derived Safety Case: Explaining the Safety Notion	59
4.3	Tier II of Derived Safety Case: Arguing over the Variables	60
4.4	Tier III of Derived Safety Case: Arguing over the Paths	62
5.1	Spacecraft Decomposition (adapted from [114, 148])	78
5.2	Arguing over Formalization of Safety Requirement	81
5.3	Arguing over Sufficiency of Safety Conditions for Individual Variables	84
6.1	High-level Architecture of Navigation System	94
6.2	Architecture Slices Recovered for Example Requirements	96
6.3	Arguing from System-Level Requirements to Component-Level Require- ments	99
6.4	Component-level safety case for Frame Conversion	101
7.1	Safety Case Templates for \wedge -Rules	113
7.2	Safety Case Templates for \vee -Rules	114
7.3	Safety Case Templates for \Rightarrow -Rules	115
7.4	Safety Case Templates for \neg -Rules	116

7.5	Safety Case Templates for \forall -Rules	116
7.6	Safety Case Templates for \exists -Rules	117
7.7	Safety Case Templates for $=$ -Rules	118
7.8	Hypothesis Handling in \Rightarrow -Rules	118
7.9	External Hypothesis	119
7.10	Generic Safety Case Template to Handle Muscadet Book Keeping Rules	120
7.11	Example of Muscadet ND Proof Safety Case	122
7.12	ND Proof Safety Case of Initialization-Before-Use	124
7.13	Abstraction of Consecutive Rules (<i>concl_only</i>)	125
7.14	Abstraction of Partial Order Reasoning Rules	126
7.15	After Abstraction: Initialization-Before Use Proofs	127
8.1	Tier I of the Heterogenous Safety Case: Explaining the Safety Notion	131
8.2	Tier II of the Heterogenous Safety Case: Arguing over Safety Property	133
8.3	Tier II of the Heterogenous Safety Case: Arguing over Safety Requirement	134
8.4	Tier II of the Heterogenous Safety Case: Arguing over Architecture Slices	135
8.5	Tier III of the Heterogenous Safety Case: Arguing over Sufficiency of Safety Condition	136
8.6	Tier IV of Derived Safety Case: Arguing over Soundness of Formal Proof	137
9.1	Safety Case Generation	143
9.2	Extended FSSC System Architecture	144
9.3	XML: Program Certification Information	145
9.4	XML: Proof Information	146
9.5	XSLT: Transforming Program Certification Information into GSN-XML	147
9.6	XSLT: Transforming Proof Information into GSN-XML	148
9.7	GSN-XML: Program Certification Information	149
9.8	GSN-XML: Proof Information	149
9.9	Code: Printing ASCE Node	150
9.10	ASCE Node Information	151
9.11	Code: Setting Link of ASCE Nodes	152
9.12	ASCE Link Information	152

List of Tables

2.1	Comparison of Safety Property and Liveness Property [23, 130]	18
2.2	Differences between FSSC and PCC	27
2.3	Terms and Definition	29
2.4	Basic Properties of Explanation [91, 92]	32
2.5	Principal Elements of the GSN (adapted from [86, 135, 155])	36
3.1	AFCG Component Groups	46
4.1	Safety Case Pattern: Explaining the Safety Notion	65
4.2	Safety Case Pattern: Explaining the Safety Notion	66
4.3	Safety Case Pattern: Explaining the Safety Notion	67
4.4	Safety Case Pattern: Explaining the Safety Notion	68
4.5	Safety Case Pattern: Explaining the Safety Notion	69
4.6	Safety Case Pattern: Safety of Program Variables	70
4.7	Safety Case Pattern: Safety of Program Variables	71
4.8	Safety Case Pattern: Safety of Program Variables	72
4.9	Safety Case Pattern: Sufficiency of Safety Condition	73
4.10	Safety Case Pattern: Sufficiency of Safety Condition	74
4.11	Safety Case Pattern: Sufficiency of Safety Condition	75
5.1	Safety Case Pattern: Formalization of Safety Requirement	87
5.2	Safety Case Pattern: Formalization of Safety Requirement	88
5.3	Safety Case Pattern: Formalization of Safety Requirement	89
5.4	Safety Case Pattern: Formalization of Safety Requirement	90
5.5	Safety Case Pattern: Formalization of Safety Requirement	91
6.1	Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements	104
6.2	Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements	105
6.3	Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements	106
6.4	Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements	107
6.5	Safety Case Pattern: Arguing Component-Level Safety Requirements to Source Code	108
6.6	Safety Case Pattern: Arguing Component-Level Safety Requirements to Source Code	109

6.7	Safety Case Pattern: Arguing Component-Level Safety Requirements to Source Code	110
7.1	Detail of Labels as used in Figure 7.12	123
9.1	Safety Case Checklist: Fault Tree Analysis (adapted from [55])	153
9.2	Safety Case Checklist: Argument Structure (adapted from [55])	154
A.1	Roles and Principles of FSSC Components [44, 47, 153]	166
A.2	Roles and Principles of FSSC Components [44, 47, 153]	167
A.3	Roles and Principles of FSSC Components [44, 47, 153]	168

Glossary

Term	Definition
Accident	Event that occurs unexpectedly and unintentionally. Here specifically as an unsafe condition (i.e., system failure) due to presence of hazards at the system-environment boundary.
Adelard ASCE v3.5	Tool for safety case construction.
Annotation	Logical expression that expresses properties that hold at a given location during the execution of a program; usually integrated into the program code as assertions of formal comments.
Annotation Inference Algorithm	Constructs annotations by analyzing the program structure, i.e., annotates the program at key program locations.
Annotation Schema	Template used by the annotation inference algorithm for the annotation construction.
Argumentation	Process of showing that some view or statement is correct or true.
AutoCert	Generator plug-in to support the formal certification of automatically generated code, based on annotation inference algorithm.
AutoFilter	Generator that automatically generates implementations for the state estimation problems.
Code Generator	Tool that translates a high-level problem specification into executable source code.
Control Flow	Transition between two statements in a program that might happen during an execution of the program; usually represent as an edge in a control flow graph.
Dataflow	Data dependency between different statement in a program; usually represent as an edge in a data flow graph.

DCM	Direction cosine matrix; describes the orientation of object relative to a coordinate reference frame.
Def-Use Chain	The underlying concept of the annotation inference algorithm, i.e., find all program locations where the safety-relevant information will be established (defined) and all potentially unsafe location (location that can violate the safety), where it is used.
ECI	Earth-Centered Inertial; describes the coordinate reference frame with its origin at center of mass of the earth.
Explanation	Process of showing why and how some phenomenon occurred or some event happened.
External Certification Assumption	Logical formula expressing assumption about a system input signals.
Failure	<p>A deviation of the system behavior from its specifications. In the context of the research prescribed here, types of failures are considered, failures of the generated program and of the formal program verification system, respectively:</p> <ul style="list-style-type: none"> • generated program - deviation from the given safety property and safety requirement • formal program verification system - inability to produce accurate proofs for the generated program
Fault Tree Analysis	Top down search method for analyzing possible faults and failures to the program safety and certification process and their interaction logic that initiate the hazard.
Formal Software Safety Certification	Process of formally proving that a program satisfies a given safety property or a set of safety requirements.
Formal Program Verification	Process of formally proving program correctness wrt. a given specification.
Goal Structuring Notation	Graphical argumentation notation which explicitly represents individual elements of a safety argument and the relationship that exist between these elements.

Generated Code	Code that is automatically generated by the code generator.
GN&C	Guidance, Navigation, and Control system; vehicle subsystem.
Hoare Logic	Set of logical rules for reasoning about the safety of a program.
Hazard	A state of a system that may lead to an accident. In the context of the research prescribed here, any conditions that could violate the given safety property and safety requirement.
Hazard Analysis	Process of identifying the characteristic of hazards, determining their significance and evaluating measures to be taken to control and mitigate the hazards.
Internal Certification Assumption	Logical formula expressing assumptions about an internal signal connecting two system components.
Muscadet Prover	Automated theorem prover based on the natural deduction calculus.
NASA	National Aeronautics and Space Administration, responsible for space program, aeronautics and aerospace research.
Natural Deduction Calculus	Proof system for the predicate calculus that is based on a set of specific rules.
Natural Deduction Proof	Sequence statements that follow the rules of the natural deduction calculus to derive a goal.
Navigation System	Vehicle subsystem, part of GN&C; designed to process location and movement data.
Nominal Behavior	Operation of the system in the event of normality.
Off-nominal Behavior	Operation of the system in the event of failures or anomaly.
Program Verification Hazard Decision Matrix	Severity classification scheme to assess the severity level caused by the disagreement of the code generator and the certification system.
Proof Tree	A representation of the underlying natural deduction proof construction that starts with the conjecture to be proven as root and the given axioms and hypotheses at each leaf of the tree.
Quaternion	Four-dimensional vector used to describe rigid body orientation in space e.g., matrices and angles.

Real-Time Workshop	Commercial code generator that automatically generates code from Simulink models.
Reference Implementation	Standard implementation of a component that can be used in place of specification.
Reference Interpretation	Standard assignment of meaning to symbols.
Safety	Freedom from accidents. Here specifically freedom from any violation of the given safety property and safety requirement.
Safety Condition	Logical formula describing the safety of a statement wrt. a given safety policy.
Safety Case	A structured argument, supported by a body of evidence that provides a valid case that a program is safe with respect to the given requirements (i.e., safety property and safety requirement).
Safety Case Abstraction	Mechanism to construct minimal but consistent safety case to reduce complexity and allow better understanding.
Safety Policy	A set of Hoare rules designed to show that safe program satisfies a safety property of interest; formalized as Hoare triples (i.e., $\{P\}C\{Q\}$), which are extended with a shadow variable and a safety predicate.
Safety Predicate	Denote semantic safety condition of the program.
Safety Property	A property stating that “something bad does never happen” during the execution of the program; formalized as an invariant that needs to hold everywhere in the program.
Safety Requirement	A property that needs to hold for the program to be safe; formalized as an assertion that needs to hold at particular location in the program (usually at the end).
Shadow Variable	Record information for the corresponding program variable that is related to the safety property.
Software Fault Tree Analysis	Top down search method for analyzing possible software logic errors and uses proof by contradiction as reasoning approach.
Velocity	Direction and speed of vehicle movement.

DECLARATION OF AUTHORSHIP

I, **Nurlida Basir**, declare that the thesis entitled **Safety Cases for the Formal Verification of Automatically Generated Code** and the work presented in the thesis are both my own, and have been generated by me as the result of my own research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- any part if this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations this thesis entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published.

Signed:

Date: *25th June 2010*

List of Publications

1. Basir, N., Denney, E. and Fischer, B. (2010) Deriving Safety Cases for Hierarchical Systems in Model-based Development. To appear in Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFE-COMP'10), 14-17 September 2010, Vienna, Austria.
2. Basir, N., Denney, E. and Fischer, B. (2009) Deriving Safety Cases from Automatically Constructed Proofs. In: Proceedings of the 4th IET International Conference on System Safety, 26-28 October 2009, London, UK.
3. Basir, N., Denney, E. and Fischer, B. (2009) Deriving Safety Cases from Machine-Generated Proofs. In: Proceedings of the Workshop on Proof-Carrying Code and Software Certification (PCC'09), 15 August 2009, Los Angeles, California, USA.
4. Basir, N., Denney, E. and Fischer, B. (2008) Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information. In: Proceedings of the 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP'08), LNCS 5219, 22-25 September 2008, Newcastle Upon Tyne, UK.
5. Basir, N., Denney, E. and Fischer, B. (2008) Deriving Safety Cases for the Formal Safety Certification of Automatically Generated Code. In: Proceedings of the International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'08), 29 March 2008, Budapest, Hungary.

Acknowledgements

I would like to thank my supervisor Dr. Bernd Fischer, for his invaluable help, guidance and encouragement. I would also like to thank Dr. Ewen Denney and Prof. Michael J. Butler whose assistance, support and advice in the last three and half year, have been invaluable and admirable, and Dr. Paul W. Garratt and Dr. Tim P. Kelly for agreeing to be the examiners to my work. My thanks also go to my dear Dean (Prof Dr. Jalani Sukaimi), my sponsors (KPT and USIM), friends and colleagues (although cannot list all names here, as every single one of them contributed to the success of the work presented) for their support and encouragement. Last but not least, I would like to thank my family, in particular my mum and dad (Al-Fatiha), for their love and support has given me so much strength.

To my dearest family, friends and relatives . . .

Chapter 1

Introduction

Chapter 1 describes the problem statement and its solution. It also explains the proposition and structure of the thesis.

1.1 Introduction

This thesis discusses issues associated with the formal program verification approach for demonstrating software safety, such as how to comprehensively show that a program satisfies all given safety requirements and safety properties based on the formal proofs, and how to establish trust in these proofs. Such issues have become a major concern in formal program verification [70, 73, 122]. Software development standards [61, 111, 113] usually describe only evidence to be produced, but the arguments and claims on how the evidence, whether informal or formal, supports the verification objective are left implicit. Hence, it is hard to tell whether the given evidence meets the objective, if no explicit explanation or arguments are provided. Recently, a goal-based argument technique known as safety cases [29, 86, 113] has been introduced to justify certain claims based on the reliable evidence [68, 126]. The development and acceptance of safety cases is now one of the key elements of safety regulation in many of safety-critical sectors [56, 99, 113, 145]. In this thesis, the safety case technique is used to argue the acceptability of formal program verification in demonstrating the safety of a program with respect to all given requirements, in particular for automatically generated code. In our work, we concentrate on constructing a safety case for automatically generated code because there is no human insight in the process to generate the code, in contrast to manual development. However, it is easy to automate the safety case construction from the generated code due to the regularities in the code. Further to this, an approach on how to automatically construct the safety cases is described.

1.2 Problem Description

Model-based design and automated code generation [40] have become popular, but substantial obstacles remain to their widespread adoption in safety-critical domains: since code generators are typically not qualified, there is no guarantee that their output is safe, and consequently the generated code still needs to be fully tested and certified [44, 108, 138, 152]. To realize the full benefits of code generation, there must be some explicit evidence, or even proofs, of the correctness of the generated code. In correct-by-construction techniques such as deductive synthesis [136] or refinement [131] this is provided by a mathematical meta-argument. However, such techniques remain difficult to implement and extend and have not found widespread application. Currently, most generators are validated primarily by testing [138, 139], as recommended by software development standards such as DO-178B [61]. However, the testing process can be time-consuming and expensive, and cannot guarantee the same level of assurance as a full verification. As a result, this can slow down generator development and deployment especially in safety-critical domains.

Formal methods such as formal program verification which are based on mathematically oriented techniques have been proposed as a means to demonstrate and improve software quality by providing formal proofs as explicit evidence for the assurance claims [37, 75, 76]. Formal program verification can be used in a product-oriented assurance approach, where checks are performed on each and every program rather than on the generator itself, showing the generated code to be correct or at least safe.

However, several problems remain. For automatically generated code it is particularly difficult to relate the proofs to the code [48, 52]; moreover, the proofs are the final stage of a complex process. In addition, these proofs are typically constructed by automated theorem provers (ATPs) based on machine-oriented calculi such as resolution [124] which are often too complex and too difficult to understand, because they spell out too many low-level details [74, 149, 150]. The proofs may be also based on assumptions that are not valid, or may contain steps that are not justified [63, 122]. This complicates an intuitive understanding of the assurance claims provided by the proofs. Moreover, it often remains unclear what is actually proven [34]. In addition, there is no traceability provided between the proofs on one side and the certified program and the used tools on the other side to gain confidence in the formal certification process. Consequently, questions such as whether the proofs really imply safety or whether the VCs are appropriate still remain as an issue. Concerns also remain on using these proofs as *evidence* or even argument in safety-critical applications [122].

Hence, it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claims and the proofs rest. In this thesis, we address these problems and present an approach to construct safety cases from information collected during a formal analysis, based on automated theorem proving, of

the automatically generated code. Safety cases [29, 86, 113] are structured arguments, supported by a body of evidence, that provide a convincing and valid case that a system is acceptably safe for a given application in a given operating environment. The Goal Structuring Notation (GSN) [86] is used as a technique to explicitly represent the structure of the safety case graphically.

To date, GSN has been applied mostly in European companies, and mainly in safety-critical industries. However in most companies, safety cases are constructed traditionally (i.e., manually) [90, 99, 145]. Manual construction is impractical especially when demonstrating the safety of large and complex software-intensive systems which requires marshalling large amounts of diverse information, e.g., models, code, specifications, mathematical formalization, and proofs. Issues such as:

- size and complexity of the constructed safety case;
- difficulties in coordinating and presenting results from many different sources and using different types of evidence;
- difficulties in finding a trace through the information provided;
- difficulties in generating and maintaining the safety case, especially when dealing with iterative software development and code generation;

should be considered when constructing the safety case manually. Obviously, tools supported by automated analyses are needed to produce a traceable safety argument [48] that shows in particular where the code, verification artifacts (e.g., proofs, verification conditions, etc.) and the argument itself depend on any external assumptions.

1.3 Outline of the Solution

Since there are many possible ways in which the trustworthiness in certification claims can be compromised, we use fault tree analysis [94] to identify the chain of causes and their interaction logic that initiate the undesired events in the formal program verification. Here, we consider undesired events as any certification failures or errors (for example, any missing and incorrect certification information or error in a certification tool) or combinations of them that might undermine the assurance provided by the formal proofs on safety of the program wrt. the given safety property and safety requirement. A safety property is a property stating that “something bad does never happen” during the execution of the program. Two types of safety properties are considered in this work, language-specific properties and domain-specific properties. Language-specific properties concern the safety aspects of the code which depend on the semantics of the programming language, while domain-specific properties concern the use of the code in

a particular domain. A safety requirement is usually related to the functional requirements of the system. It has usually been identified during the hazard analysis of the overall system. In our work, we assume the safety requirements as given. Any possible failure in the system wrt. the safety requirements should be controlled and mitigated in order to prevent a failure from turning into an accident. In our work, we use the fault tree analysis as a guideline in the safety case construction. We have constructed four different types of safety cases that argue about the program safety wrt. the given safety property and the given safety requirement respectively, about the architecture slices of the system, and about the soundness of the formal proofs. We show in the safety cases how all the potential undesired events have been controlled and perhaps mitigated. We provide evidence to justify our claims.

The core argument structure of our safety cases is based on information collected by a formal analysis (called annotation inference) of the generated code [46, 47]. It is driven by a set of requirements (safety properties and safety requirements) and assumptions of the program. In particular, it analyzes the system structure on the code level to identify where the requirements are ultimately established, and so checks the program, providing independent assurance. It also identifies how the system safety requirements are broken down into component requirements, and where they are ultimately established, thus establishing a hierarchy of requirements that is aligned with the hierarchical model structure. The overall argument structure is then extended by auxiliary verification and validation information, which is separately specified. In addition, we also present an approach to develop safety cases that correspond to formal proofs found by automated theorem provers. Here, we concentrate on natural deduction style proofs which are closer to human reasoning than for example resolution proofs.

The resulting safety cases will make explicit the formal and informal reasoning principles, and reveal the top-level assumptions and external dependencies that must be taken into account in showing software safety. The safety cases thus provide a “structured reading guide” for the software and the safety proofs that will allow users to understand the safety claims without having to understand all the technical details of the formal machinery and also provide a traceable route to the safety requirements, safety claims and evidence that are required to demonstrate software safety. We use the GSN [86] as a technique to explicitly represent the logical flow and linkage of the safety argument in the safety case.

In our work, instead of manual safety case construction, we present an approach to systematically (and ultimately automatically) derive a safety case from formal program verification information. So far we have automatically constructed safety cases for safety properties, safety requirements, hierarchical system structures and safety proofs. We leave the implementation that integrates diverse types of other verification and validation information such as testing result with the existing constructed safety cases for future work.

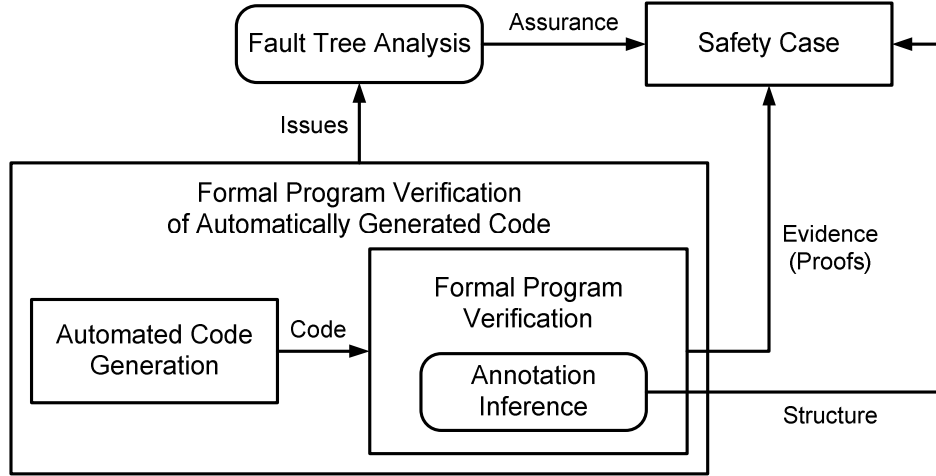


FIGURE 1.1: Approach: From Formal Proofs to Safety Cases

Figure 1.1 shows an overview of our automatic safety case generation approach. All doubts concerning the correctness of the proofs, the correctness of any of the certification tools and whether the proofs actually entail program safety are analyzed using a fault tree analysis technique. Our main focus is on process safety i.e., identifying the events that can induce the hazard related to the underlying proofs construction. Def Stan 00-56 requires that the safety of the product is argued in a separate safety case [112, 135]. Here, safety cases are then used to provide a defensible argument that all of these doubts have been controlled and thus, establish trust why the generated code can be assumed to be sufficiently safe. Here, the core argument structure of the safety cases is derived from information collected during a formal verification of the code, in particular from the construction of the logical annotations necessary for a formal, Hoare-style safety certification. The annotations formalize auxiliary safety information of the program. In our work, the structure of the program-specific argument directly follows the course the annotation construction takes through the code. In addition, the information for the remaining arguments of safety cases are provided by other verification and validation activities such as testing and background knowledge of the formal verification phase.

In principle, our approach is independent of the given requirements and program, and also independent of the underlying code generator. So far, our approach has been successfully applied to the AutoFilter [153] and MathWorks Real-Time Workshop [52] code generators, based on the information provided by the formal software safety certification system.

1.4 Thesis Proposition

This thesis demonstrates an approach to establish trust in the assurance provided by the formal proofs of safety of program with respect to a given set of requirements (i.e., safety properties and safety requirements), in particular for automatically generated code. It identifies problems and doubts associated with the formal program verification approach that can invalidate the assurance provided. Further to this, a defensible argument to reason that all undesired events have been controlled and mitigated is developed. This argument provides a traceability between the proofs, the certified program and the used tools in the formal program verification to establish trust in the safety of the program. An approach on how to automatically construct these arguments in the form of safety cases from a formal analysis, based on automated theorem proving, of the automatically generated code is also introduced.

1.5 Thesis Structure

This introduction has, so far, outlined the problem addressed by this thesis, the solution and the proposition of the research. The rest of the chapters are organized as follows:

Chapter 2, *Background*, describes relevant literature on automated code generation and current recommended approaches and standards for program certification e.g., formal methods and testing. Chapter 2 also discusses relevant literature on hazard analysis, such as techniques used in identifying and analyzing hazards. In addition, it also includes an explanation about the safety case and its applications.

Chapter 3, *Hazard Analysis for Formal Program Verification*, describes the results of a hazard analysis (i.e., fault tree analysis) for the formal program verification system. All possible events and their interaction logic that might invalidate the safety claim construction during the formal program verification phases are discussed in detail in this chapter.

Chapter 4, *Property-Oriented Safety Cases*, presents the structure of the safety case that focuses on demonstrating safety of the generated code with respect to a given safety property by providing formal proofs as explicit evidence for the assurance claims. This chapter illustrates the approach on code generated by the AutoFilter system [153] for the attitude estimation of a deep space probe.

Chapter 5, *Requirement-Oriented Safety Cases* describes an approach on how to construct safety case from formal safety requirements which express as logical formulas that a (software sub-) system's output signals must satisfy for the (overall) system to be safe. This chapter illustrates the work using the verification of two safety requirements for a

spacecraft navigation system that was generated from a Simulink model by Real-Time Workshop [4].

Chapter 6, *Architecture-Oriented Safety Cases*, describes an approach to systematically derive safety cases that argue along the hierarchical structure of systems in model-based development. This chapter illustrates the approach on flight code generated from hierarchical Simulink models by Real-Time Workshop for NASA's Project Constellation uses Real-Time Workshop for its Guidance, Navigation, and Control (GN&C) systems [114, 148].

Chapter 7, *Proof-Oriented Safety Cases*, presents an approach to develop safety cases that correspond to formal proofs found by automated theorem provers and reveal their underlying argumentation structure and top-level assumptions. In particular it shows how the approach can be applied to the proofs found by the Muscadet prover [118].

Chapter 8, *Heterogenous Safety Cases*, describes an approach on how the results from a formal analysis and additional verification and validation information such as testing results for control software can be communicated in a single integrated safety case.

Chapter 9, *Implementation and Preliminary Assessment of the Approach*, shows the feasibility of the approach (i.e., the construction of safety cases from the formal program verification information of the automatically generated code) on industrial applications. Further to this, a checklist is described on whether the constructed safety cases provide a convincing and valid argument that the program is adequately safe for a given context in a given environment based on the formal analysis information.

Chapter 10, *Conclusions*, concludes the contributions of the work described in this thesis and describes how the work differ from others. It also outlines the limitations of the approach and describes some suggestion for future work.

Chapter 2

Background

Chapter 2 describes the background literature relevant to the thesis. This chapter is divided into the following sections:

- *Automated Code Generation* describes the literature on automated code generation and gives examples.
- *Generator Assurance* describes existing approaches to guarantee the correctness of generators or the code they guarantee.
- *Formal Methods and Formal Logic* presents the literature on formal methods, in particular the difference between a formal specification and formal verification and also the difference between various types of formal logic (e.g., temporal logic and Hoare-logic).
- *Hoare-style Verification Systems* describes relevant literature on existing systems following the Hoare-style program verification approach, in particular, proof carrying code and formal software safety certification.
- *Hazard Analysis* describes the relevant literature on terminologies, models and techniques that are used for hazard analysis.
- *Argumentation* explains the differences between an argumentation and explanation approach. This section also describes a relevant literature on safety cases including safety case patterns and safety case assurances.

2.1 Automated Code Generation

Automated code generation [40], also known as program synthesis, is a technique to automatically translate a high-level problem specification into source code for the targeted platform. The high-level problem specification can be:

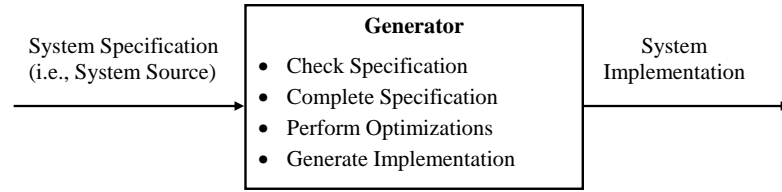


FIGURE 2.1: Main Tasks of a Code Generator [40]

- A set of formal or informal requirements of the program. Formal requirements can be written using tools such as SpecTRM [14]; informal requirements are often written in stylized natural language with the help of standard text-processing applications.
- A behavioral model of the program, written in formal specification languages like Z [133] and B [19]; this can also be specified using mathematical structures and functions such as algebraic specification [127].
- A structural model of the program, e.g., in form of a UML class diagram [16].
- A mathematical model of the underlying system, such as a set of differential equations. This can also be given in form of an executable MATLAB model [4].

Code generation has a significant potential to eliminate coding errors, improve code consistency, reduce development costs, and speed-up development. Code generators are widely used in web applications [1, 3, 5], databases applications [2, 6] and system control applications in avionics, automobiles, railways and other industries [4, 17, 153]. Code generators typically work by adapting and instantiating pre-defined code fragments for (parts of) the problem specification, and composing these partial solutions. In order to translate specifications into executable source code, a number of steps are required. The generator first reads and analyzes the input specifications and checks their validity. Valid specifications are then compiled and translated into code fragments. The generator will then check whether the code fragments satisfy the specifications and report any warnings or errors, if necessary. If there is no error, the code fragments will then be assembled, optimized and translated into an executable source code, for example C and Java. Large-scale generators that implement complex specification languages are often divided into a set of smaller and cooperating generators in order to speed up the code generation process [40]. Figure 2.1 shows the translation process of a generator.

Various code generation approaches have been introduced. The following list describes the existing approaches of code generation and their examples.

- *Code-based or generative code generation*, which is also known as a template-driven code generation. In the template-driven approach, the code templates can be used

or even modified to automatically generate the code. CodeSmith Code Generation [1] and Code-gen with Velocity [15] are examples of a template-driven source code generators. CodeSmith automates the creation of source code for languages such as C, Java, VB and SQL by using the templates designed for a specific program in a specific language. These templates can easily be modified or written based on user requirements [1]. Code-gen with Velocity automates the creation of Java program from the templates written in Velocity Template Language (VTL). Velocity uses normal Java classes, associated to a specific context, as the data model (i.e., generates more code from an existing program) and produces output in the format specified by the VTL [15].

- *Model-based or transformative code generation.* In the transformative approach, specifications are normally designed as a model by using a modeling tool. A generator then repeatedly transforms the models and translates the final model into the target languages, e.g., C or Ada. Real-Time Workshop Embedded Coder 5.1 [4] is an example of commercial model-based generator that generates executable C code from Simulink and Stateflow models. The executable code is ANSI/ISO C compliant, enabling it to run on any microprocessor or real-time operating system (RTOS). Another example of a model-based code generator is SCADE Suite KCG - DO178B Code Generator [17, 32]. KCG produces C code from the SCADE model that has all properties required for safety-critical embedded software. Similar to Real-Time Workshop, KCG also produces executable code that is ANSI C compliant and works on any target microprocessor. Our focus in this thesis is on model-based code generators because it is the most common approach in safety-critical application domains.
- *Proof-based or deductive code generation.* In the deductive approach, the specification is defined as logical theorem to be proven and transformations of the specification into the program can be represented as inference rules. Kestrel Interactive Development System (KIDS) [131] is an example of proof-based code generator that supports the development of correct and efficient programs in various domains, including scheduling, combinatorial design, graph theory, and linear programming. KIDS provides automated tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation and data type refinement. The purpose of KIDS tool is to provide correct and efficient executable code with less effort. However KIDS has been applied to a few domains only and is difficult to use in practice. In addition, since we do not get any insight from the way of the proofs been constructed in KIDS, the application of our approach is quite difficult.

However, some generators use a combination of these three approaches to automatically generate code. For example, they may generate and translate code into a target platform by repeated application of schemas (template-based approach), use a mathematical

model (e.g., statistical models and differential equations) as input in the code generation (model-based approach) and check the applicability of the templates by theorem proving (proof-based approach). Below are examples of the generators that are based on mixed approaches:

- CTADEL [54] is a generator of multi-platform high performance codes for partial differential equation-based scientific applications that is applied to the HIRLAM numerical weather forecast system.
- The AutoFilter [153] Program Synthesis System is a program synthesis system that takes high level specifications of state estimation tasks and derives code by repeated application of schemas, i.e., algorithms and solution methods of the domain. This synthesis system automatically generates and translates code into a target platform, for instance C and Modula-2, after schemas are applied that use the appropriate variants of the Kalman filter.
- The AutoBayes [59] Program Synthesis System is a fully automatic program synthesis system for the statistical data analysis domain. AutoBayes has been applied to a number of advanced textbook examples, machine learning benchmarks, and NASA applications. It is based on the same underlying approach as the AutoFilter synthesis system which was also developed at NASA Ames.

Nowadays, *model-based development* has become popular in software engineering. It comprises a number of techniques that focus on creating and transforming domain-specific abstractions or *models* rather than algorithmic concepts or even code. It has offered improved productivity and simplified development of large systems as the possibility to generate code automatically from the model accelerates the software development process. In *model-based design* [4, 128], mathematical or, more commonly, visual methods are used to create an initial model of the system design. It is commonly used in the control systems domain, where block diagrams provide an accepted notation. Blocks can represent arbitrary computations and can be nested hierarchically, which helps countering system complexity. They are connected by wires that represent the flow of signals through the system. A large number of academic and commercial tools support model-based design in this domain [4, 8, 17, 153]. Many NASA projects use it for at least some of their modeling and code development, particularly in the GN&C domain, where MathWorks Simulink [4] is commonly used. Simulink comes with a large library of standard modeling blocks that provide mathematical operations and signal routing suitable for control systems and complex operations.

Model-based code generation [109, 128] usually complements model-based design, and translates specifications in the form of a model into a program in a high-level programming language such as C or Ada. The translation process is often organized as a sequence of model transformations, where the last model is equivalent to the program. The final

source code generation step can then be implemented using a simple template engine such as CodeSmith Code Generation [1]. In this thesis, we focus on one commercial generator, Mathworks Real-Time Workshop Embedded Coder 5.1 [4]. Real-Time Workshop generates ANSI/ISO compliant C and C++ code from MathWorks Simulink and Stateflow models. Embedded Coder is an add-on that has various additional features which are useful for generating C code tuned for embedded devices.

2.2 Generator Assurance

Although automated code generation has become commercially available, substantial obstacles remain to its widespread adoption in safety-critical domains [108], which in turn lead to missing assurance for the correctness and safety of the generated code. Generator assurance is thus substantially important, especially in safety-critical applications. However, generator assurance as required by software development and certification standards, for example DO-178B [61], requires enormous effort and is difficult to accomplish. Most of the standards are process-based, and provide extensive guidelines for measuring safety and quality of the applied software development tools. For example, DO-178B has defined a guideline for the use of code generators in the avionics industry. DO-178B requires the development process of the code generator itself to satisfy the same objectives as the development process of other airborne software. The standard also requires that the generator is assured as safe and does not initiate or contribute to a failure in the aircraft's functions. Basically, DO-178B requires the submission of a valid justification as to why and how code generator achieves the required certification goals. Obviously, to realize any benefits from code generation, the generated code thus needs to be shown correct or at least safe. In correct-by-construction techniques such as deductive synthesis [136] or refinement [131] this is done by a mathematical meta-argument. However, such techniques remain difficult to implement and have not found widespread application. In addition, the meta-argument does not provide any insight in the generated code.

Most previous work on generator assurance has focused on techniques to ensure correctness of the code generator via testing. Stürmer and Conrad [138, 139] present a systematic testing approach and safeguarding techniques for model-based code generator. Safeguarding refers to techniques and procedures which are applied to increase confidence in the generated code as well as to ensure that the generator works as expected. The technique provides a guideline for manually reviewing the generated code, testing the generator and conducting a generator simulation. Stürmer and Conrad [138] also introduce a test suite approach as a means to identify missing requirements and irregular behavior caused by improper formal specifications. Similarly, SCADE [32] introduces an approach for the verification of the generator based on requirements-based testing. SCADE uses the SCADE simulator and Design Verifier for the verification of

the source code by SCADE Suite KCG - DO178B Code Generator [17, 32]. The development of SCADE KCG follows a step-by-step approach to satisfy the objectives of safety standards, i.e., DO-178B Level A standard for software applications in airborne systems. The requirements-based testing processes verify that SCADE KCG complies with the system requirements. For example, Stephenson *et al.* [135] develop an argument to assess the development of SCADE KCG adequately meets the objectives of Def Stan 00-56. However, the testing efforts easily become excessive and time-consuming, and testing on its own is insufficient to provide enough assurance, especially in safety-critical systems. Moreover, testing is usually done for a specific generator. Thus, the testing results are not valid and cannot guarantee the same assurance for any new version of the tool, and the entire testing process must be repeated.

Full functional correctness of the generator, which would imply correctness and safety of the generated code, is difficult to show practically, due to the generator's complexity, size and frequent changes. Similar issues also arise in compiler verification [78], which is similarly hard to achieve in practice because of the complex architecture, sophisticated analysis and optimization algorithm that are used in compilers. However, Necula suggested that *if we cannot prove that a compiler is always correct, then maybe we can at least check the correctness of each compilation* [107]. This observation has inspired the technique of translation validation [107, 121, 159], which checks the result of each compilation against the source program and detects any compilation errors. The translation validation approach offers assurance of the correctness of the machine code by providing formal proofs showing that the machine code is a correct implementation of the source program. In translation validation, the code is validated thoroughly on each and every compilation process. It is a promising technique to isolate compilation errors and consequently, to increase the reliability of compilers. Since compilation and automated code generation are based on similar concepts, the same approach can also be applied for a code generator. Instead of assuring correctness of the code generator, checks can be performed on each and every generated program.

Thus, many researchers believe that a product-oriented assurance approach [44, 66, 76, 108] such as translation validation is a viable alternative to testing. Here, assurance is not implied by the trust in the generator but follows from an explicit arguments for the generated code. Jones and Glenstrup [85] argue that in order to establish trust in the generated programs, a firm semantic basis is needed and must be clearly and precisely defined in order to guarantee that users' intentions match the behavior of the generated programs. Jones and Glenstrup also suggest that in order to make clear what was specified in the model, first, the semantics of the specification language must be clearly understood. Second, evidence such as proofs and testing results are needed to show that the outputs of the program generator have the same semantics as specified by its inputs or specifications. Third, the program generator and the programs that it generates must adhere to particular software standards. These suggestions are also

in line with what has been described by Whalen and Heimdahl in [151, 152]. Whalen and Heimdahl describe a minimum set of requirements for creating code generators that are fit for application in safety-critical systems. In addition to Jones and Glenstrup, Whalen and Heimdahl add an additional requirement for code generation assurance, i.e., the generated code must be well structured, well documented and easily traceable to the original specifications. Clearly, it can be seen that both approaches argue that the assurance should be ultimately derived from the code rather than from the generator.

In addition, O'Halloran [108] and Denney and Fischer [44] suggest explicit evidence through formal proofs should be provided to show the correctness of the generated code. Denney and Fischer demonstrate their ideas in the certifiable program generation approach [43, 44, 49]. It uses automatic program verification and is based on four basic ideas. Firstly, in order to assure the safety of the code, the generated program itself is certified, not the generator. Secondly, the generator needs to be extended to support the construction of the necessary logical annotations together with the generated code. Here, annotations record all pertinent information that is necessary for an automated theorem prover to prove that the program is safe. Thirdly, the proof is based on partial correctness rather than total correctness of the program, and Hoare-style program verification (cf. Section 2.3.2 for details) is used as the underlying verification approach. Hoare-style program verification provides a rigorous mathematical foundation for producing safety proofs that illustrate partial correctness of the generated program. Finally, the objective of certifiable program generation is to show that the generated code conforms to safety-relevant aspects identified, i.e., a given *safety property*. The safety property serves as a high-level requirement of the program assurance. It is an exact characterization of a property stating that *something bad never happens* [23, 119, 130] during the execution of the program. The safety property is defined based on the operational semantics of the programming language. Different safety properties are formalized by different *safety policies*. Safety policies are formalized as a set of Hoare rules to show that the generated program satisfy the safety property of interest. As usual in Hoare-style program verification, a verification condition generator (VCG) will turn the annotated program into a set of verification conditions (VCs) and if all VCs are proven to hold by an automated theorem prover (ATP), the program can be concluded to be safe with regard to the given safety property.

2.3 Formal Methods and Formal Logic

2.3.1 Formal Methods

In recent years, formal methods which are based on mathematical and logical techniques, have been proposed as a means to improve software quality. Formal methods [37, 73, 74] are applicable to the specification, design, and verification of software and computer

hardware. A rigorous formal methods application helps in better understanding complex systems, eliminating ambiguity of natural language descriptions and, producing software that is correct. There are two main parts to a formal method [37], formal specification and formal verification. Formal specification is a process of describing the system and its desired property using a mathematically defined syntax and semantics. Formal specification techniques offer a deeper understanding of the system being specified. Several examples of formal specification languages and methods are Z [133], B [19] and temporal logic [119, 120].

Formal verification on the other hand is a process of proving correctness of software or computer hardware with respect to formal specification. Two well-established approaches to formal verification are model checking and theorem proving. Such techniques have significant potential to ensure correctness and reduce cost in software and hardware verification. Model checking [37, 116] is a technique to verify that a desired property holds in a finite model of a system. Checks are performed on all possible states that a system could enter during its execution. Model checking helps to tell why a property is not satisfied (i.e., it produces counterexamples of the claimed property). It also helps in finding bugs in the program, which will help in the debugging process. Theorem proving [37, 116] conversely is a process of finding a proof of a property from given axioms and rules. The proof of a property is then a means to establish trust on safety of the program. Traditionally, formal program verification concentrated on showing full functional equivalence between the specification and the program, but recent applications are more on showing safety and security aspects that can be defined and formalized via a safety property. In this research, we use the formal program verification of the automatically generated code as basis for the construction of safety cases.

2.3.2 Formal Logic

Predicate Logic. Predicate logic [60, 83] is used as a means to overcome the limitations of propositional logic in encoding the declarative sentences. Propositional logic usually deals with sentence components, such as *not*, *and*, *or* and *if...then* without covering the logical aspects of other natural languages such as *all* and *only*, for example, the declarative sentence [83]:

Not all birds can fly.

Propositional logic is unable to represent the properties in this sentence and to express their logical relationships, dependences and truth. In contrast, predicate logic can be used to communicate the relation that exist among the objects and properties in this sentence. Predicate logic (also referred to as first-order logic) represents a relation between these properties that can be true or false. Two properties are identified for the sentence i.e., x is a bird and x can fly. We use variable x to replace the bird's name,

and quantifiers \forall and \exists to describe the meaning of *all* and *there exist* in the sentence. The sentence can be represented as:

$$\neg(\forall x(B(x) \rightarrow F(x)))$$

Alternatively, we could rephrase the above sentence as *It is not the case that all things which are birds can fly*, which gives the same meaning. We could represent this as:

$$\exists x(B(x) \wedge \neg F(x))$$

In our work, we use formal proofs found by ATP as evidence on safety of the program. ATPs usually use formal logics such as first-order logic, type theory, higher order and modal logics to express a specification of arbitrary problems. In proving correctness and safety of a program, different types of proofs (such as resolution proofs and natural deduction proofs) are provided by theorem provers which will be described in the next subsection.

Liveness and Safety Properties. Temporal logic [23, 119, 120, 130] provides a formalism for specifying and verifying correctness of computer programs. It is a mechanism to relate the states of a system over time. It is appropriate for reasoning over sequential and non-terminating concurrent programs. Temporal logic gives rise to the notions of safety property and liveness property. Table 2.1 shows a comparison between safety and liveness property.

In temporal logic, the correctness of a program can be shown based on two concepts, invariance and eventuality [119]. Invariance is used to prove that a safety property holds continuously throughout the execution of a program and eventuality is used to prove both the correct behavior in time non-terminating programs and liveness properties. Basically, the notion of eventuality also includes proofs of total correctness. Alpern and Schneider [23] show that every property of a program is an intersection of a safety property and a liveness property. Moreover, total correctness is defined as a combination of partial correctness and termination.

In our work, we focus on partial correctness proofs, i.e., by showing that the program does not violate certain conditions during its execution, and establishes certain other conditions if and when it terminates. Our work is based on a notion of safety property that is slightly more specialized than the general notion of safety property as defined in temporal logics. We consider two types of safety properties, language-specific properties and domain-specific properties. Language-specific properties concern the safety aspects of the code which depend on the semantics of the programming language, while domain-specific properties concern the use of the code in a particular domain. However, both versions are safety properties in the sense that they are invariants that the program is not allowed to violate during its execution. We also consider safety requirements in

TABLE 2.1: Comparison of Safety Property and Liveness Property [23, 130]

Safety Property	Liveness Property
Informal definition: Something bad never happens during program execution	Informal definition: Something good will eventually happen during program execution
A safety property corresponds to partial correctness which does not ensure termination, but only that all terminating computations produce correct results	Liveness property corresponds to total correctness which guarantees termination
Proof method based on global invariants extensively used for proving correctness with respect to the safety property	Proof method based on proof lattices or well-founded induction extensively used for proving correctness with respect to the liveness property

addition to the safety properties, which are assertions that need to hold at particular occurrence in the program and thus have more of a liveness “flavor”.

Hoare Logic. Hoare logic [77] is extensively used for proving correctness of a program wrt. a correctness specification, e.g., a safety property. In Hoare logic, the correctness of a program can be specified in two ways:

- A partial correctness specification of the form $\{P\}C\{Q\}$ says that whenever P holds for the state before the execution of C , and C terminates then Q holds afterwards.
- A total correctness specification of the form $[P] C [Q]$ says that if P holds before the execution of C , then C is guaranteed to terminate, and when it does, then Q holds.

The relationship between partial and total correctness can be expressed as:

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}$$

A central feature of Hoare logic is the Hoare triple. A triple describes how the execution of a command changes the state of the computation. For partial correctness, a Hoare triple is in a form:

$$\{P\}C\{Q\}$$

where P is a precondition, Q is a postcondition and C is a command. Both precondition and postcondition are assertions. Assertions are formulas in predicate logic. Essentially, there are two ways to read the triple, forward and backward [22, 44]. In the forward direction, the triple can be read as *whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which C s execution terminates satisfies Q* . In the backward direction, the triple can be read as *in order to establish Q after executing C , P must hold*. In addition, in the forward direction, in order for Q to be valid,

Q has to follow from the strongest postcondition $\text{spc}(C, P)$ of C with respect to P , while in backward direction, in order for P to be valid, P has to entail weakest precondition $\text{wpc}(C, Q)$ of C with respect to Q .

Hoare logic provides axioms and inference rules for each statement type in the programming language, for instance while, if-then-else and assignments. Axioms like the assignment axiom:

$$\{P[X/E]\}X := E\{P\}$$

describe valid triples, while inference rules like the composition rule:

$$\frac{\{P\}C_1\{R\}, \{R\}C_2\{Q\}}{\{P\}C_1; C_2\{Q\}}$$

describe how to derive valid triples from other given valid triples. For example, let us consider the Hoare triple [22]:

$$\{x = 5\}x := x * 2\{x > 0\}$$

If we read this triple in forward direction, we can see that it is clearly correct: if $x = 5$ and we multiply it by 2, we get $x = 10$ which clearly implies that $x > 0$. However, although correct, this Hoare triple does not use the strongest postcondition. For reasoning in forward direction, we would prefer a stronger postcondition instead of any valid postcondition. For example, $x > 5 \wedge x < 20$ is stronger because it is more informative and it pins down the value of x more precisely than $x > 0$. The strongest postcondition is $x = 10$. Formally, if $\{P\}C\{Q\}$ and Q is the strongest postcondition of C with respect to P , then Q logically implies all other post-conditions that make the triple valid.

In another example, let us consider the while program [22] as below:

```

r := 1;
i := 0;
while i < m do
  r := r * n;
  i := i + 1

```

The proof rule for while loops uses a *loop invariant* P , i.e., a formula that is true at the begin and the end of the loop's body:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\text{while } B \text{ do } S \text{ done } \{\neg B \wedge P\}}$$

We need to determine the right loop invariant as different loop invariants could result in different preconditions. Based on the program, we must have $m \geq 0$ in order to enter the while loop and we can assume $n > 0$. Therefore we can define the precondition as

$m \geq 0 \wedge n > 0$ and the loop invariant as $r = n^i$. However, this is not strongest loop invariant, as we need to know that $i = m$ as exit condition. Therefore we can add $i \leq m$ to the loop invariant together with $0 \leq i$, $m \geq 0$ and $n > 0$ in the precondition to give a complete invariant for proving the loop body is correct. Thus the full loop invariant will be $r = n^i \wedge 0 \leq i \leq m \wedge n > 0$.

Several logics have been introduced as extensions to Hoare logic. For example, O’Hearn *et al.* [110] introduce *separation logic* for reasoning about programs that manipulate pointer data structures. Owicki and Gries [117] extend Hoare’s system for the verification of parallel programs with shared variables. Jones [84] introduces the rely-guarantee method, a compositional version of the Owicki-Gries system. Our work primarily focuses on Hoare logic as a means for assuring correctness of a program wrt. all given requirements (i.e., safety properties and safety requirements).

2.3.3 Formal Proofs

Natural Deduction. A formal proof is a sequence of statements that follows certain rules of reasoning. Most automated theorem provers produce proofs that are often complex and difficult to understand, because the rules of reasoning used to construct and encode them is machine-oriented e.g., resolution [134]. In resolution proofs, the conclusion is proved following the proof by contradiction technique (if the falsity has been shown impossible, then the conclusion must be true) [25, 79, 134] and using only very few inference rules. Machine-generated proofs may also be based on assumptions that are not justified. This causes concerns about the trustworthiness of using them as arguments in safety-critical applications. Therefore, various efforts [25, 79] have been made to transform such proofs into a more readable presentation that is closer to the human reasoning approach, in particular into natural deduction proofs.

Natural deduction [64, 79, 83] is an attempt to provide a foundational yet intuitive system to construct formal proofs. It consists of a collection of proof rules that manipulate logical formulas and transform premises into conclusions. A conjecture is proven from a set of assumptions if a repeated application of the rules can establish it as conclusion. The proof rules can be divided into basic rules, derived rules (which can be seen as proof “macros” that group together multiple inference steps) and replacement rules (which are derived rules for equivalence and equality handling).

Natural deduction uses two sets of rules for each logical connective or quantifier (e.g., $\wedge, \vee, \Rightarrow, \forall$), where one introduces the symbol, while the other eliminates it. In the introduction rules, the connective or quantifier is used as the top-level operator symbol of the unique conclusion, while it occurs in the introduction rules in the same role in one of the premises. For example, the rule for conjunction introduction concludes that if A is true and B is true, then evidently $A \wedge B$ is true as well (\wedge -i), while conjunction

elimination says if $A \wedge B$ is true, then both A and B must be true as well (\wedge - e_1 resp. \wedge - e_2). The conjunction rules can be written as:

$$\frac{A \quad B}{A \wedge B} \wedge\text{-}i \quad \frac{A \wedge B}{A} \wedge\text{-}e_1 \quad \frac{A \wedge B}{B} \wedge\text{-}e_2 \quad (2.1)$$

Further discussion related to natural deduction rules can be found in Chapter 7 of this thesis. A full exposition of natural deduction can be found in the literature [83].

Proof Verbalization and Visualization. Other approaches have been used to address concerns with using proofs for assurance purposes. Many of them also try to bring formal proofs into a form closer to human reasoning, to aid with their understanding. Proof visualization tools (e.g., [142]) present the proof in a graphical form, but quickly get overwhelmed by the proof size. Proof verbalization (e.g., [36, 80]) transforms the proofs into natural language but the explanations are often too detailed. Proof abstraction groups multiple low-level steps that represent recurring argumentation patterns into individual abstract steps and thus accentuates the hierarchical structure of the proof [51] but has so far only been applied to interactively constructed proofs. Our work combines abstraction, verbalization and visualization to reveal and present the proofs' underlying argumentation structures and top-level assumptions.

Proof Checking. Proof checkers [101, 140] have been used to increase trust in formal proofs, by demonstrating that every individual step in the proof is correct. However, proof checking does not address the real problem: while errors in the implementations of provers do occur, they are very rare [63]; errors and inconsistencies in the formalization of the domain theory in contrast are much more common [21, 125], but these are not detected by the standard proof checking techniques.

2.4 Hoare-Style Verification Systems

This section presents and compares two examples of Hoare-style proof systems that have been used to demonstrate the partial correctness of a program wrt. a given safety property. We focus on implementations that are integrated into compilers and code generators respectively.

2.4.1 Proof-Carrying Code

Proof-Carrying Code (PCC) [106] is a method for ensuring safe execution of untrusted machine code. It is based on Hoare logic, which is used to formally demonstrate that the code satisfies the safety property of interest. The general presentation of PCC is centered on the interaction between a code consumer, a code producer and a proof validator.

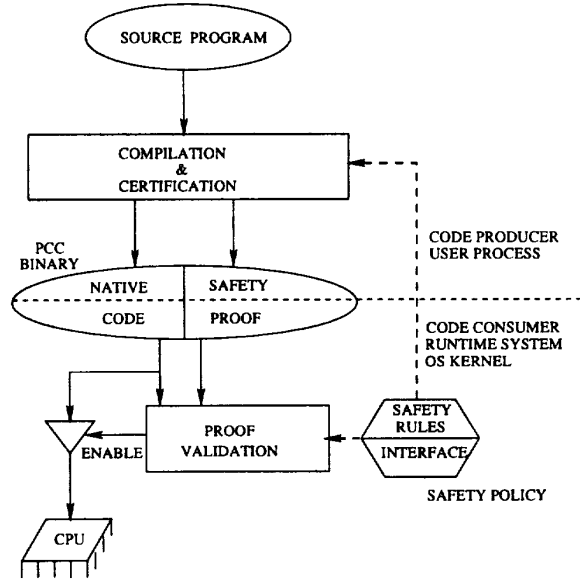


FIGURE 2.2: Overview of Proof-Carrying Code [106]

The code producer is required to establish safety proofs that prove the untrusted code's adherence to the formalized safety policy. The proof validator is required to verify and double-check the validity of the proofs generated. The code consumer will then check, with the help of proof validator, that the safety proofs are valid and hence guarantee that the untrusted code is safe to execute wrt. the given safety policy. Code producer and code consumer must agree on a safety policy that formalizes the safety property of interest, and will guide the verification process. A safety policy consists of two main components, namely the safety rules and the interface. The safety rules describe all legal operations and safety preconditions of the code, while the interfaces describe essential conventions between the code consumer and the untrusted code. Figure 2.2 shows an overview of the PCC approach.

According to Necula [106], an implementation of proof-carrying code must consist of the following five elements:

- a formal specification language used to express the safety policy;
- a formal semantics of the language (i.e., in the form of a logic) used by the untrusted code;
- a language that allows natural representation of the defined logic in order to express the proofs;
- an algorithm for validating the proofs; and
- a method for generating the safety proofs.

Proof-carrying code is based on the same Hoare-style program verification techniques as the verification approach described in this thesis. Section 2.4.3 shows the comparison between the approaches.

2.4.2 Formal Software Safety Certification

Formal software safety certification (FSSC) [43, 46, 47] is a technique for ensuring safety of automatically generated code. FSSC is a product-based assurance approach, where rather than verifying and validating the generator, checks are performed on each and every generated program. FSSC uses formal source code analysis techniques based on program logics to show that the program does not violate certain conditions during its execution (i.e., meets the given safety property) and thus remains safe during execution. A safety property is an exact characterization of certain safety conditions, based on the operational semantics of the programming language. It thus serves as high-level requirement for program assurance. In FSSC, there are two types of safety properties, namely language specific properties (e.g., initialization-before-use safety and absence of out-of-bounds array accesses) and domain-specific properties (e.g., matrix symmetry and coordinate frame consistency). Different safety properties are formalized by different safety policies in a form of Hoare logic.

Each safety policy is a set of Hoare rules designed to show that safe programs satisfy a safety property of interest. A safety policy is essential in maintaining a logical safety environment and producing the appropriate safety obligations of the program. It is formalized using the usual Hoare triples (i.e., $\{P\}C\{Q\}$), which are extended with a shadow variable and a safety predicate [43]. A shadow variable records information for the corresponding program variable that is related to the safety property, and a safety predicate denotes semantic safety conditions of the program. The set of all shadow variables constitute the logical safety environment, such as shown in the example below [44]:

$\frac{}{P [E/X] \{ X:=E \} Q}$	$\frac{}{P [E/X, \text{INIT} / X_{\text{init}}] \wedge \text{safe}_{\text{init}} \{ X:=E \} Q}$
a) Standard assignment rule	b) Assignment rule for initialization safety

FIGURE 2.3: Assignment Rules

In order to achieve a fully automated verification, a program logic requires annotations (i.e., preconditions and post-conditions, and loop invariants) at key program locations. Annotations are assertions (i.e., logical formulae) that express desired condition at various intermediate points. These annotations serve as lemmas that facilitate the proof of the verification conditions (VCs) that are produced from the annotated program. VCs are produced by a verification condition generator (VCG) which traverses the annotated

code backwards and applies the safety policy to produce VCs. If all VCs are proven by an ATP, it can be concluded that the program is safe with regard to the given safety property. In its current form [43, 46, 47], FSSC only works on the intermediate representation of the source code and provides only partial correctness proofs (no termination). It also does not support any real-time or floating-point reasoning.

Figure 2.4 shows the overall system architecture of the FSSC. Here, the code generator is complemented by the annotation inference subsystem and the standard Hoare-style verification components (i.e., VCG, automated theorem prover and proof checker) to achieve a fully automated verification of the generated code. Several tools such as simplifier, TPTP2X-converter, and domain theory are also used to help in the automatic verification. The simplifier simplifies the VCs before they are transmitted to the ATP, the TPTP2X-converter converts VCs into the particular first order logic syntax of the different ATPs and a domain theory supports the capability of ATPs in providing proofs. The domain theory consists of fixed axioms, lemmas, and rewrite rules. In FSSC, the components are distinguished into two categories, namely trusted (in pink) and untrusted components (in light blue). Trusted components are crucial to the assurance and must be correct because any error in them can compromise the assurance provided by the overall system, while untrusted components are not crucial to the assurance as their results can be checked by at least one trusted component. The FSSC follows the certifiable code generation approach as introduced in [43, 44, 49]. FSSC system shifts the trust burden from the code generator and the program to the certification system: instead of having to trust an arbitrary generated program to be safe, users have to trust the certifier to be correct.

Annotation Inference. A program logic requires annotations at key program locations to perform a fully automated verification. This can in principle be achieved by integrating into the generator to support for the construction of the annotations, in particular by embedding annotation templates into the code templates, which are then instantiated and refined in parallel by the generator. However, this requires a tight integration of code generator and annotation construction. Alternatively, annotations can be constructed by a separate annotation inference algorithm that is completely independent of the code generator and analyzes the generated code after the fact.

The key technical idea of the approach is to exploit the idiomatic nature of auto-generated code in order to automatically infer the annotations. The idea of the annotation inference algorithm [46, 47] is to “get the information from definitions to uses”, i.e., to find all program locations where the safety-relevant information will be established or “defined”, as well as all potentially unsafe locations, where it is used, and then to construct the formulae required for the annotations, and to annotate the program along the control flow-paths between definitions and uses. The annotations along the control-flow paths formalize the property that needs to be maintained by the program for the use to be safe. The annotations will then be converted into three sets of VCs

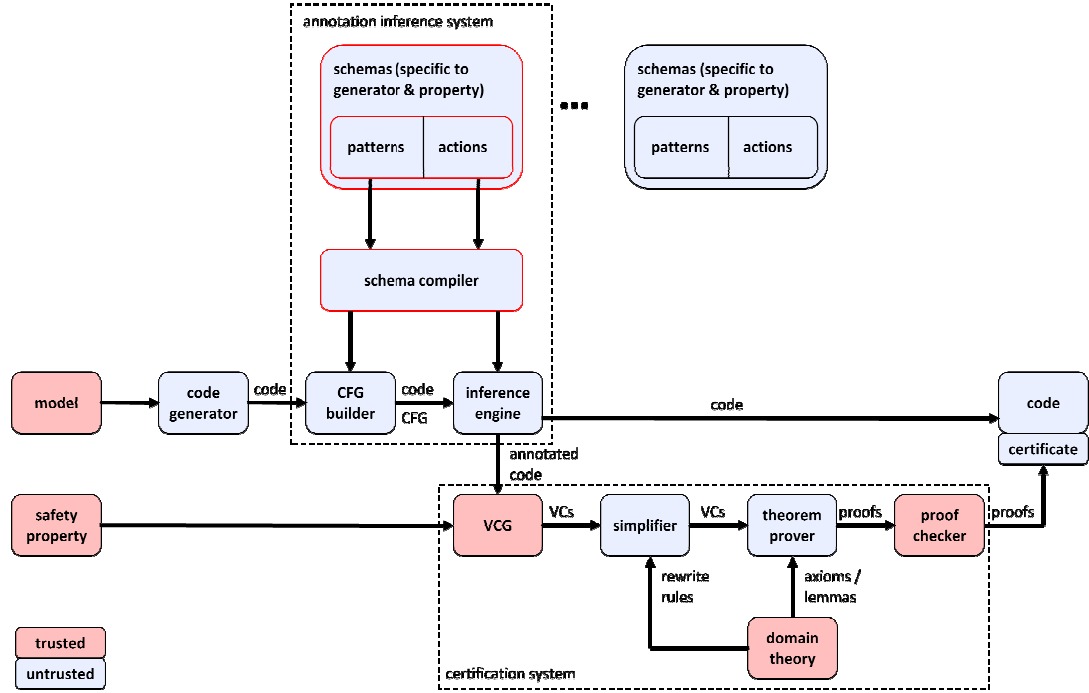


FIGURE 2.4: Architecture of Formal Software Safety Certification System [44]

by the VCG, corresponding to establishing and maintaining appropriate invariants, and satisfying the safety condition in the program; if and only if all VCs can be shown to hold, then safety property holds for the entire program. Note that an annotation only serves as an auxiliary lemma that contains the desired safe conditions of the program; any error in an annotation does not compromise the assurance provided but only leads to unprovable VCs.

The annotation inference algorithm is generic, and parameterized with respect to a library of annotation schemas that depend on the safety policy and the code generator. The schemas consist of a code pattern and a set of actions. The patterns characterize the notions of “definitions” and “uses” that are specific to the given safety property. For example, for initialization-before-use safety property, definitions correspond to variable initializations while uses are statements which read a variable, whereas for array bounds safety, definitions are the array declarations, while uses are statements which access an array variable. The actions are used to compute the annotations that are inserted into the code when the schema is applied. A schema compiler translates the high level declarative schemas into low-level term manipulations.

AutoCert: Automatic Certificate Generation. AutoCert [52] is a generator plugin to support the subsequent certification of the code created by the AutoFilter Program Synthesis Tool [153] and MathWorks Real-Time Workshop [52] code generators. AutoCert is based on an implementation of the generic annotation inference algorithm described above. It has been instantiated for a range of mathematically oriented re-

quirements, mostly in the vehicle navigation domain. Given a set of formal assumptions (e.g., constraints on input signals) and requirements (e.g., constraints on output signals), it formally verifies that the generated code complies with the specified requirements. It also supports the certification of several safety properties, as described in Section 2.4.2. This provides high-level assurance about the safety and reliability of the code without excessive manual verification and validation effort.

Since AutoCert follows the FSSC approach to certification, it carries out a symbolic analysis of the generated source code in order to infer the required annotations and ultimately to prove properties about the code. During the course of this analysis (where it effectively “reverse engineers” the code), AutoCert records various facts, such as the locations of variable definitions and uses, which are later used as an input in the safety case generation process.

2.4.3 Proof-Carrying Code vs Formal Software Safety Certification

FSSC uses a similar approach as PCC for ensuring safety of a program i.e., Hoare-style program verification for the given safety property. Both of these approaches use formal proofs to demonstrate that the program adherence to the given safety property (e.g., type safety and memory safety), rather than showing full functional correctness of the program. They are also similar in:

- the components that are used in the verification framework, i.e., VCG, ATP, proof checker, safety property and safety policy;
- the category of components which are distinguished into two categories, namely trusted and untrusted components;
- the verification process (i.e., it starts with identifying and constructing a safety policy to represent the safety property of interest, and continues with instantiation of annotations that formalize safety of the program, followed by formally verifying the VCs by ATP and finally providing formal proofs for the assurance claim on program safety);
- the certification assurance provided, i.e., provides formal safety proofs as assurance on safety of the program with respect to the given safety property.

However, there are some aspects that are different in these two approaches. FSSC is focused on auto-generated code, while PCC is focussed on mobile code such as JavaScript and ActiveX controls. In addition to the PCC, FSSC can ensure not only the safety of program wrt. the language specific property (e.g., initialization-before-use and array bounds safety) but also wrt. the properties and requirements that relate to the program’s

TABLE 2.2: Differences between FSSC and PCC

	FSSC	PCC
Domain	Automatically Generated Code	Mobile Code
Property	Language specific property Domain specific property	Language specific property
Code Level	Source Code	Object Code

application domain (e.g., matrix symmetry). Both FSSC and PCC also work on different level of code representation: while FSSC works on an intermediate representation of the source code, PCC works on the object code level. Table 2.2 shows these difference aspects of FSSC and PCC. In our work, we use FSSC as it works on the source code level which is closer to human readable language than object code.

2.5 Hazard Analysis

2.5.1 Hazard Analysis Terminology

As computers became increasingly important components of complex systems, concerns about the correctness and safety of the software began to emerge, especially in safety-critical systems. Research has shown that software failures have led to catastrophic accidents as for example in the Mars Climate Orbiter Crash [20] and Therac-25 accidents [98]. According to the accident report, Mars Climate Orbiter crashed in September 1999 because of a “silly mistake”, i.e., the use of wrong units in a program. In another case, a large number of cancer patients died due to overdoses of radiation resulting from a race condition between concurrent tasks in the Therac-25 software [98]. It can be seen that software safety [94, 132] is not merely a software-specific issue but it also a part of the system safety issues.

Since software is a part of a system, the hazards caused by a software must be analyzed and mitigated in order to retain system safety. Hazard analysis [94] is a process of identifying the characteristic of hazards, determining their significance and evaluating measures to be taken to control and mitigate the hazards. Hazard analysis is a tool within the discipline of system safety engineering and “at the heart of an effective safety program” [94].

In system and software safety, terms are used inconsistently, where different literature interprets similar terms in slightly different meanings. Frequently, confusion exists between the notions of failure, error and fault, and sometimes they are used interchangeably. Leveson [94] defines *failure* as an “event or a behavior that occurs at a particular

instant of time”, while an *error* is a “static condition or a state that remains until it has been removed”. In comparing fault and failure, Leveson describes *failures* as “basic abnormal occurrences”, for example a burned-out bearing in a pump or a short circuit in an amplifier and *faults* as “abnormal conditions or defects at the component or sub-system level”, for instance an improper functioning of some upstream component. In general, Leveson concludes that “all failures are faults, but not all faults are failures”. In the same review, Leveson also points out the association between hazard and accident, where a *hazard* is defined as a “potential condition that can cause harm to personnel, system, property or environment”, while an *accident* is an “event that occurs unexpectedly and unintentionally”. For example, if an accident is defined as a collision between two aircraft, then a possible hazard is the lack of minimum separation between aircraft. “The longer the hazardous state exists, the greater the chance of the accident to occur, thus the higher the risk” [94].

Table 2.3 provides definitions of the basic terms that are frequently used in hazard analysis. In our work, we customize the definition of the terms such as hazard, error, fault, failure and safety to suit the research context (see Chapter 3 Section 3.1).

2.5.2 Hazard Severity Classification Scheme

Different certification standards classify hazard severity in different schemes. MIL-STD-882 [111] defines a severity scheme with four categories, namely catastrophic, critical, marginal and negligible, while the FAA System Engineering Council (SEC) [132] severity scheme defines five classes, namely catastrophic, hazardous, major, minor and no safety effect. The severity classification is used as an indication of the worst possible effect caused by the identified hazard in the particular environment. The severity classification scheme is important as a guideline in analyzing the hazards of a critical system, especially when human life is involved. In a safety-critical system, safety experts are responsible to identify the effect of each associated system hazard. The experts are required to classify each of the effect in terms of its severity, from the most severe to the least severe, by referring to the severity classification scheme. They also have to identify the likelihood of the occurrence of the hazard, i.e., either it anticipated to occur frequently, probably, occasionally, remote or improbable.

Since software failures have become one of the main contributors to catastrophic accidents (see for example [20, 95, 98, 115]), the assessment of the severity level of software failures and their effects to the environment is crucial. In our work, we customize the severity scheme (see Chapter 3 Section 3.3) to fit our context in analyzing errors and faults in program verification process that can cause doubt on whether the proofs really entail program safety and cause failure to the software and hence a system.

TABLE 2.3: Terms and Definition

Term	Literature Definitions
Error	<ul style="list-style-type: none"> - Design flaw or deviation from a desired or intended state [94]. - A defective value in an erroneous state of a system [82].
Fault	<ul style="list-style-type: none"> - Defect in part of a component or in the design of a system [82]. - A manifestation of an error in software. A fault, if it occurs, may cause a failure [32].
Failure	<ul style="list-style-type: none"> - Inability of a system or component to perform its intended function for a specified time under specified environmental conditions [94]. - A deviation of the system behaviour from its specification [82]. - Inability of any component of the system to perform its intended function or to perform it correctly within specified limits [93].
Hazard	<ul style="list-style-type: none"> - State or set of conditions of a system, together with other conditions in the environment system that will lead to an accident [94]. - A state of a system with the potential for harm [82]. - Any condition, event, or circumstance, which could induce an accident. A potentially unsafe condition. A situation which has the potential to lead to harm [93]. - A physical situation or state of a system, often following from some initiating event, that may lead to an accident [113].
Safety	<ul style="list-style-type: none"> - Freedom from accidents or losses [94]. - Freedom from unacceptable risk [93].
Accident	<ul style="list-style-type: none"> - Undesired, unavoidable and unplanned (but not necessarily unexpected) event that results in (at least) a specified level of loss [94]. - An unintended event, or sequence of events, that causes harm or, also defined as a mishap [113].

2.5.3 Hazard Analysis Models and Techniques

There are a number of established models and techniques for identifying hazards and assessing their probabilities and severities. Some differ primarily in their names, whereas others truly have their own unique and important characteristics. In the development of safety-critical systems, functional hazard assessment, fault tree analysis and failure mode effect analysis techniques are extensively used for hazard analysis. The techniques help in identifying and analyzing the hazards and also in preparing a plan to cope with the identified and unidentified hazards.

Functional Hazard Assessment (FHA) [105, 132, 143] is a “systematic and comprehensive assessment method for identifying the potential functional failures or hazards of each component in the system” [105, 143]. FHA starts with defining the functionality of each component, analyzing the consequences of their failure and assigning their severity level. Results of the analysis are documented in a tabular listing diagram. Appendix A shows the roles and principles of the formal program verification components used in FSSC. However, the further analysis of the consequences of each component’s failure and the corresponding severity levels is not part of this thesis.

Fault Tree Analysis (FTA) [12, 94, 143] is “a top-down search method used for analyzing causes of hazards, but not for identifying the hazards” [94]. FTA is a “procedure for determining the various combinations of hardware and software failures and also human errors that could result in the occurrence of specified undesired events at the system level” [12]. FTA begins with a single undesirable event, then attempts to work backward and forward to identify and determine the chain of events and their interaction logic that initiated the most undesirable event. FTA uses Boolean logic to illustrate the combinations of these undesired events. Results of the analysis are presented graphically as a tree structure and built using conventional logic gate symbols with the undesirable event at the root and the cause event at the bottom of the tree. The undesired events are connected to each other using ‘OR’ and ‘AND’ gate symbols. In our work, we use FTA to analyze possible events that might invalidate the safety claim construction. Details of the analysis are discussed in the next chapter.

Software Fault Tree Analysis (SFTA) [94, 96, 97] is slightly different from fault tree analysis, as it focuses more on software safety (e.g., program errors) than on system safety [94]. SFTA is suitable for “verifying the safety aspects of software” [94, 96, 97]. The analysis starts with the software hazard that could affect systems safety, and traces backward to find paths through the code to detect software logic errors. SFTA uses mathematical proof by contradiction as reasoning approach. It starts with the hypothesis that the software has produced an unsafe output, and further down the analysis shows that this event could not happen because the hypothesis leads to a contradiction. SFTA is only concerned with the “safety of the code logic” [94, 96, 97]. In SFTA, the code is analyzed in isolation from the system analysis. SFTA uses Boolean logic and trees

to illustrate the chain of the undesired events graphically. The root of the tree is the software hazard, while the necessary preconditions that initiated the defined hazard are described at the leaves of the tree. Nodes are connected with either a logical 'OR' or a logical 'AND' relationship. However, the application of SFTA is less significant here since we have deferred the argument on the correctness of the VCG to future work.

Failure Mode Effect Analysis (FMEA) [94, 132, 143] is a “reliability analysis that focuses on assuring successful system operation” [94]. The main objective of this analysis is to identify actions and strategies to mitigate and control the potential failure modes and their effect on the system operations. The results of this analysis are presented in a table with columns that include details about component, failure probability, failure mode, percent failures by mode, and their effect. FMEA documents all possible failures of a system, determines the effect of each failure on system operation, and ranks each failure according to its severity. By anticipating these failure modes in the software and system development, it helps in identifying actions to overcome the identified failure and enhancing the reliability of the system. FMEA also can be used to check the completeness of the resulting fault tree by analyzing the effects of each single failure at the bottom of the tree and assessing the seriousness of this failure mode to the overall system. However, we left the implementation of FMEA for future work.

2.6 Argumentation

2.6.1 Explanation and Argumentation

Explanatory information is essential to increase understandability and eliminate misperception. Explanation [58, 92] constitutes an “intelligent dialogue” that is understandable and uses natural language as medium of its interaction. Lacave and Diez [91, 92] describe several basic properties of explanations as shown in Table 2.4.

Generally, explanation is based on three main concepts: content, communication and adaptation. The most important aspect is the content of the explanation, i.e., what is going to be explained and how clear it could be. The content should contain the focus, purpose, level, and causality of the explanation to be presented. The communication is concerned with how the explanation is going to be offered. In a computer system, the explanation can for example be delivered by a natural language dialogue, by selecting options from a menu, or by predefined questions. The explanation can be displayed or presented in text, graphically, or interactively via multimedia tools such as video and audio. The final aspect is adaptation or to whom the explanation is offered. Variation of users' knowledge, users' expectation, and the level of detail of the explanation are properties that should be considered in making a good explanation. In a heuristic expert systems, Lacave and Diez [92] divided explanation methods into:

TABLE 2.4: Basic Properties of Explanation [91, 92]

Category	Property	Options
Content	- Focus - Purpose - Level - Causality	- evidence/model/reasoning - description/comprehension - micro/macro - causal/non-causal
Communication	- User-System Interaction - Display of Explanations	- menu/predefined questions/ natural language dialogue - text/graphics/multimedia
Adaptation	- Users knowledge about the domain - Users knowledge about the reasoning method - Level of Detail	- no model/scale/dynamic model - no model/scale/dynamic model - fixed/threshold/auto

- *Non-adaptive methods*: The detail of explanation is fixed. The interaction between the system and the user is made through predefined questions or options, for example in text planning [33, 102].
- *Adaptive methods*: The detail of explanation is not fixed. The interaction between the system and the user is conducted dynamically where explanations are addressed in different level based on the user request. Examples are reactive explanations [102, 103] and dialog planning [33].

Frequently, confusion exists between the notion of explanation and argumentation. Argumentation [104] is a medium for generating interactive and collaborative explanations. From a theoretical point of view, explanation and argumentation are two notions that are relatively close and difficult to distinguish [141]. The main difference between them is that in explanation the topic to be explained is not discussed whereas in argumentation the topic is explicitly argued [144]. Hughes [81] defines the purpose of an explanation is to show why and how some phenomenon occurred or some event happened, while the purpose of the argument is to show that some view or statement is correct or true. Explanation is appropriate when we are seeking to understand why something occurred, while argument is appropriate when we want to show that something is true, usually when there is some possibility of disagreement about its correctness. Lacave and Diez [92] define two types of argumentation as follows:

- *Textual argumentation*, where arguments are presented in free text using natural language. A user has the ability to request an explanation and the system will then generate the new explanation with strong argumentation in natural language.
- *Graphical argumentation*, where graphical notations are used to clearly communicate the relationship between the claim and evidence. Research has shown that

graphical representation can improve human ability to solve ill-structured real world problems in comparison to textual representation. Kelly [86], in his research showed that textual argumentation can lead to unclear and ambiguous argumentation due to poorly structured English.

In addition, Schroeder [129] claims that visualization of argumentation is a promising approach to make complex reasoning of single or multiple agents more intuitively understandable, without requiring knowledge of the foundations of logic. Given the motivation to show that “something bad never happens during the execution of the program”, argumentation seems to be the appropriate approach for this research in comparison to the explanation approach. Here, the argument helps in showing the truth of the assurance claim provided by the certification system. The argument will also help to establish trust and confidence on the assurance provided.

2.6.2 Safety Cases

Safety cases [13, 29, 86, 113] are “structured arguments, supported by a body of evidence, that provide a convincing and valid case that a system is safe for a given application in a given operating environment” [29, 113]. Both Bishop and Bloomfield [29] and Kelly [86] have defined the main elements of a safety case. Bishop and Bloomfield [29] decompose a safety case into the following four main elements:

- *Claim* i.e., the property of the system or some subsystem;
- *Evidence* i.e., the basis of the safety argument. Evidence can be facts, assumptions, or subclaims;
- *Argument* i.e., a link of the evidence to the claim. Arguments can be deterministic, probabilistic or qualitative; and
- *Inference* i.e., the mechanism that provides the transformational rules for the argument.

Figure 2.5 illustrates the relationship that exists between the elements as suggested by Bishop and Bloomfield. Similarly, Kelly [86] points out that a safety case consists of four key elements as follows:

- *Requirements* i.e., the safety objectives that must be addressed to assure safety;
- *Evidence* i.e., the information from analysis, testing or simulation of the system;
- *Argument* i.e., a link showing how the evidence indicates compliance with the requirements; and

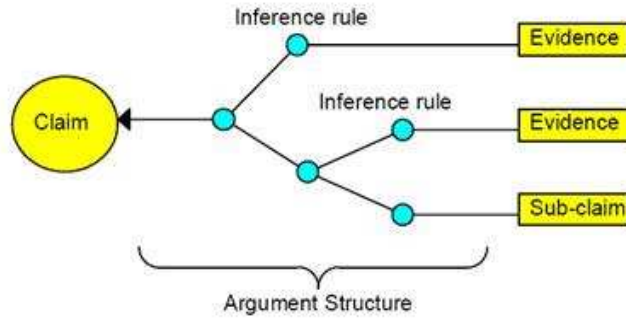


FIGURE 2.5: Principal Elements of Safety Case - Bishop and Bloomfield [29]

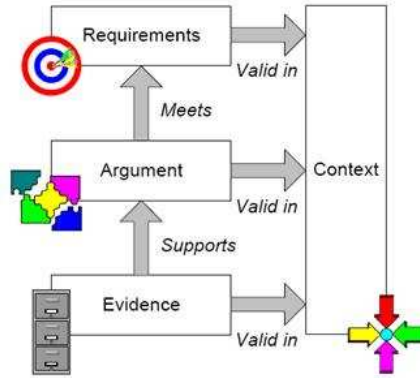


FIGURE 2.6: Principal Elements of Safety Case - Kelly [86]

- *Context* i.e., the basis of the overall argument presented in the safety case.

Figure 2.6 illustrates the *macro-dependencies* that exist between the elements suggested by Kelly [86]. There are significant commonalities between the safety case elements defined in [29] and [86]. Both approaches agree that evidence and argument are important elements in a safety case. Kelly also claims that “an argument without evidence is unfounded and evidence with no argument is unexplained” [86]. However, Kelly uses term requirement instead of claim to describe the target to be met by the system or subsystem. Despite the similarities, Bishop and Bloomfield [29] introduced three types of inference rule as a means to support the link between the evidence and claim, i.e., *deterministic* - relying upon axioms, logic and proof, *probabilistic* - relying upon probabilities and statistical analysis, and *qualitative* - relying upon adherence to the standards, design code, etc, while Kelly [86] uses context to describe the contextual information of the argument such as context, constraints, justifications and assumptions to support the overall safety argument.

2.6.3 Goal Structuring Notation

Various approaches [29, 30, 57, 86] have been developed and introduced to construct safety cases. In most recent studies, the Goal Structuring Notation (GSN) has been used to model the entities and relationships as shown in Figure 2.6. GSN [65, 86, 89, 155] is a graphical argumentation notation which explicitly represents individual elements of a safety argument and the relationships that exist between these elements. GSN also offers [86]:


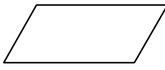

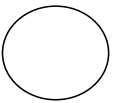
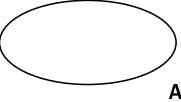
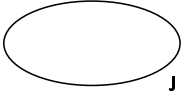
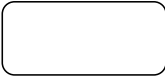
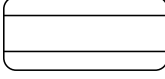


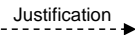
- explicit representation of the logical flow of the safety argument;
- explicit representation of the role of evidence; and
- explicit representation of the rationale underlying an argument.

Table 2.5 describes principal elements of the GSN. In this thesis, we make some modifications to the notations (see Table 2.5) to represent our arguments.

Kelly [86] describes six structured steps for the GSN construction as illustrated in Figure 2.7. Step 1 in the goal structure development is to state the goal of the argument. For example, the goal is to demonstrate that the generated code is safe to execute with respect to a safety property of interest. Having identified a goal in Step 1, Step 2 of the process requires the context of the argument to be identified. For instance, the context explains the informal interpretation of key notions like safe and safety property. Step 3 of the method requires that an argument strategy for supporting the top level goal is identified. The strategy can vary as it depends on the objective of the argument. For example, the strategy can be based on showing that all of the operating hazards have been controlled and mitigated. Then, Step 4 requires that all the underlying assumptions and judgements that support the evidence and strategy to derive the top level goal have been addressed. Further to this, in Step 5, the defined strategy has to be elaborated until the arguments are completed. Finally, in Step 6, the valid evidence to support the argument should be provided. The evidence can be the analysis results, audit reports, testing results, or any supporting documents.

Figure 2.8 shows an example of a goal structure for the Safety of Air Navigation project [56]. Here, the argument starts with the claim (G0) that the risk of collision under a Reduced Vertical Separation Minimum (RVSM) is tolerable. Cr0001 describes the requirements for the RVSM, i.e., collision risk under RVSM meets TLS for vertical risk and the overall risk is not increased. It is based on the assumption (A0001) that the current level of RVSM risk is tolerable. This is justified by the fact that RVSM is being introduced to meet a legitimate business need. The further arguments are reduced to four principal safety arguments (G1 to G4), which together represent the necessary and sufficient conditions for G0 to be true.

TABLE 2.5: Principal Elements of the GSN (adapted from [86, 135, 155])

Element	Description	Notation
Goal	Requirement, target or constraint to be met by the system or to be shown true or false.	
Strategy	Adds further detail to goal decomposition and describes how claim is addressed by the other sub-claims or subgoals.	
Model	Representation of the system, design information documentation, architectural model or process description.	
Solution	Provides backing or evidence to show that a requirement has been met. Evidence can be any related documents, analysis, results of audit reports or testing results.	
Assumption	Statement whose validity has to be relied upon in order to make an argument. Necessary in the decomposition and translation of requirement or when stating a goal or adopting a strategy.	
Justification	Rationale for the use of particular goal or strategy. Used to reason why the defined strategies are needed and why the evidence is relevant and should be believed to be true, or how the evidence and strategies can derive the stated goal.	
Context	Information necessary for an argument to be understood. Associated with the goal, strategy and solution elements. The relationship between context and these elements is labeled as "In-context-of".	
Constraint	Restriction imposed in which goal can be achieved or solved [155].	
In-context-of	Provides contextual links to constraint the scope of an element (e.g., link from goal to context, assumption and justification).	
Is-solved-by	Shows the main flow of the argument from topmost goal down to evidence (e.g., link from goal to strategy or/and solution).	
Is-justified-by	Provides links to the relevant argument or claim to justify the validity of the information.	

To date, GSN has been applied mostly in European companies, and mainly in safety-critical industries, such as:

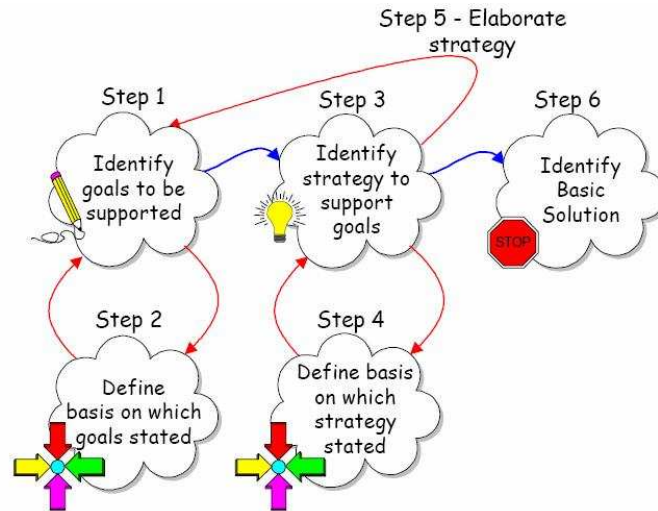


FIGURE 2.7: The Steps of the GSN Construction Method [86]

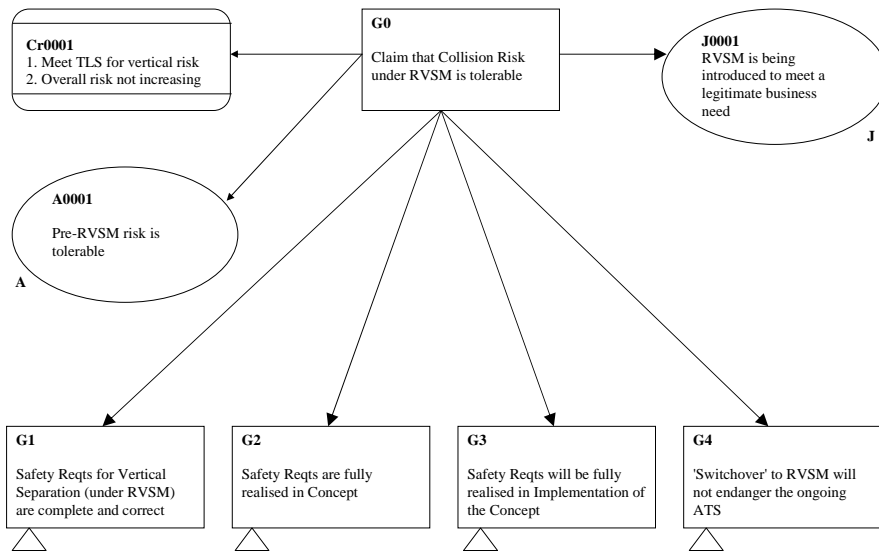


FIGURE 2.8: The EUR RVSM Pre-Implementation Safety Case - Overall Argument (adapted from [56])

- air traffic control including EUROCONTROL Airspace System [56], and
- transportation industry, for instance at London Underground Limited [99].

A variety of tools are available to support the development of safety cases, including GSN: ASCE v3.5 from Adalard [9], ISCaDE from RCM2 [13], GSN CaseMaker from ERA Technology [10] and E-Safety Case from Praxis HIS [11].

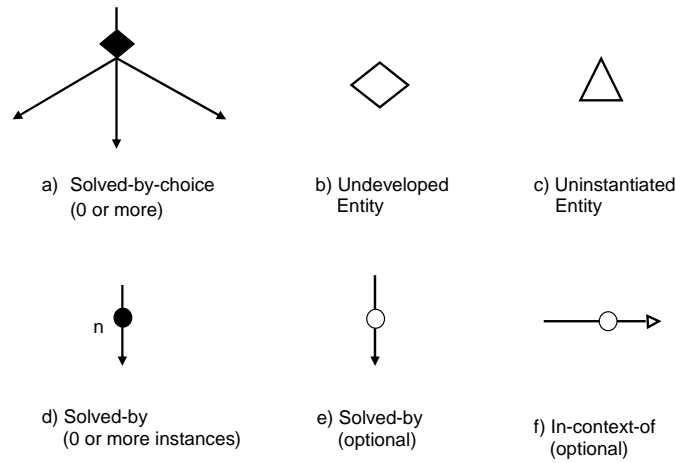


FIGURE 2.9: GSN Extensions to Represent Patterns and Their Relationships

2.6.4 Safety Case Patterns

Safety case patterns have been described as a means of documenting and reusing successful safety arguments [86, 88]. The patterns are not intended to provide a reusable model of a complete safety case but only to describe partial solutions, i.e., tackling just one aspect of the overall structure of the safety argument contained within a safety case. Safety case patterns generalize the structure of the safety argument and can be used as a way of abstracting the fundamental strategies from the details of particular safety case designs. Patterns are based upon reusable goal structures that can be instantiated to aid the construction of parts of a safety argument [146].

Alternative documentation structures have been proposed and used for the description of the safety case pattern. Kelly [86] has defined an extension of the GSN to describe a safety case pattern, i.e., generalize the details of the argument. Figure 2.9 shows the GSN extensions to represent patterns and their relationship, and Figure 2.10 shows the template for the safety pattern document. According to [86], the details of the element descriptions in the pattern document are as follows:

- **Pattern Name.** Communicates the key principle or central argument being presented by the safety argument pattern. It is important to give the right pattern name as it will be the label to identify the pattern.
- **Intent.** Describes what the pattern is trying to achieve or its purposes.
- **Also Known As.** Other names that can be used to appropriately describe the pattern.
- **Motivation.** Describes why the pattern was constructed.

Pattern Name			
Author			
Created		Last Modified	
Intent			
Also Known As			
Motivation			
Structure			
Participants			
Collaborations			
Applicability			
Consequences			
Implementation			
Example			
Known Uses			
Related Patterns			

FIGURE 2.10: Template for the Safety Pattern Document

- **Structure.** Presents the structure of the argument pattern.
- **Participants.** Describes each of the elements of the goal structure pattern.
- **Collaborations.** Describes how the different elements of the pattern work together to achieve an effective argument. It also explicitly describes the links between elements that are not communicated by the argument structure.
- **Applicability.** Describes under what circumstances the argument can and should be applied. Any assumptions and principles underlying the argument pattern should clearly described too.
- **Consequences.** Describes what work remains after carried out the argument pattern.
- **Implementation.** Communicates how the application of the pattern should be carried out and hints or techniques to ease the application of the pattern. Any possible problems in applying the pattern and misinterpretations of the terms or concepts used should be described here as well.
- **Examples.** Provides examples that illustrate the instantiation of the pattern.

- **Known Uses.** Describes known uses of the form of argument presented in the pattern.
- **Related Patterns.** Identifies any related safety case patterns.

2.6.5 Safety Case Assurance

Safety case assurance [87] is the process of evaluating the safety case to obtain mutual acceptance of the claim-argument-evidence presented. To conclude that the claim made in the safety case deserves sufficient confidence and is convincing, the argument has to be well-structured, the right evidence has to be used, the assumptions have to be accepted and the reasoning has to be consistent. Littlewood [100] introduces an approach to formally treat the argument structures by presenting a formal probabilistic treatment of “confidence” by using a Bayesian Belief Networks (BBN) model to measure the “strength” of arguments. In addition, Strigini [137] suggests the potential of BBNs to audit strengths and weaknesses of the argument. Cyra and Gorski [39] introduce an appraisal mechanism based on the Dempster-Shaffer Model of gathering expert opinions about basic elements of the argument (i.e., the value of evidence, assumptions and facts). The mechanism presents an aggregation of the opinions to assess quality of the overall argument. In different studies, Kelly [86], Despotou [53], Fan Ye [158] and Weaver [146] implement peer review, questionnaire, and case study techniques to evaluate the *feasibility* of the safety case constructed. In addition, Kelly [87] also introduces a structured approach for the argument review. He introduces four reviewing process steps:

- *Argument Comprehension*: make the argument readable, e.g., highlight the difference of argument elements (claims, assumptions, contexts, etc);
- *Well-formedness* (syntax checks): identify structural errors, e.g., claims without supporting evidence;
- *Expressive Sufficiency Checks*: identify any missing context, justifications, assumptions, constraints;
- *Argument Criticism and Defeat*: recognize the distinction between deductive and inductive arguments and audit the integrity of evidence based on several attributes such as dependency, directness, and robustness.

In this thesis, we evaluate the comprehensiveness, well-formedness and expressiveness of the resulting safety cases by using safety case checklists and by presenting them to the research community. Further intensive evaluation is left for future work.

2.7 Conclusions

This chapter presented a review of relevant literature on automated code generation and existing approaches and standards for program verification and validation, e.g., formal methods and testing. The main focus is on describing the Hoare-style formal program verification approach, which is used for ensuring safe execution of code, in particular for automatically generated code. It also discussed relevant literature on techniques used in identifying and analyzing hazards, including an explanation on safety-related terminologies. A basic overview on the differences between argumentation and explanation approach is also given. In addition, relevant literature on safety cases, the Goal Structuring Notation and its applications in safety-critical industries are discussed. Finally, the documentation structures of the safety case pattern and the approaches used for the safety case assurance are discussed in this chapter.

Chapter 3

Hazard Analysis for Formal Program Verification

Chapter 3 describes the hazard analysis methodology and the results from the fault tree analysis of the formal program verification method. Any errors or faults in the program verification process that can cause doubts whether the proofs really entail program safety are described in this chapter.

3.1 Introduction

In our work, safety is defined as freedom from any violation of the requirements (i.e., safety properties and safety requirements) imposed on the program. We use formal program verification to demonstrate this notion of safety. In this chapter, we describe the technique to identify undesired events in the formal program verification process that might invalidate the assurance claim provided by the formal proofs, i.e., errors or faults in the program verification process that can lead to invalid or incorrect proofs.

Incorrect proofs might lead us to believe that an unsafe program is safe to be executed. The execution of an unsafe program, especially in safety-critical applications, has considerably high potential to lead to a failure of the system that executes the program and thus potentially to cause an accident. This is because when a program violates a safety property or safety requirement during its execution, its results can become undetermined, (e.g., for a violation of the initialization-before-use safety property), and its interaction with the system can lead to system failure. Hence, incorrect proof can indirectly increase the overall risk of the system failure, because they provide a false sense of safety.

Since we do not know at program generation time the full system context in which the program is executed, we have to assume that any violation of the safety property or safety requirement can lead to a system failure that can cause an accident. Hence, a

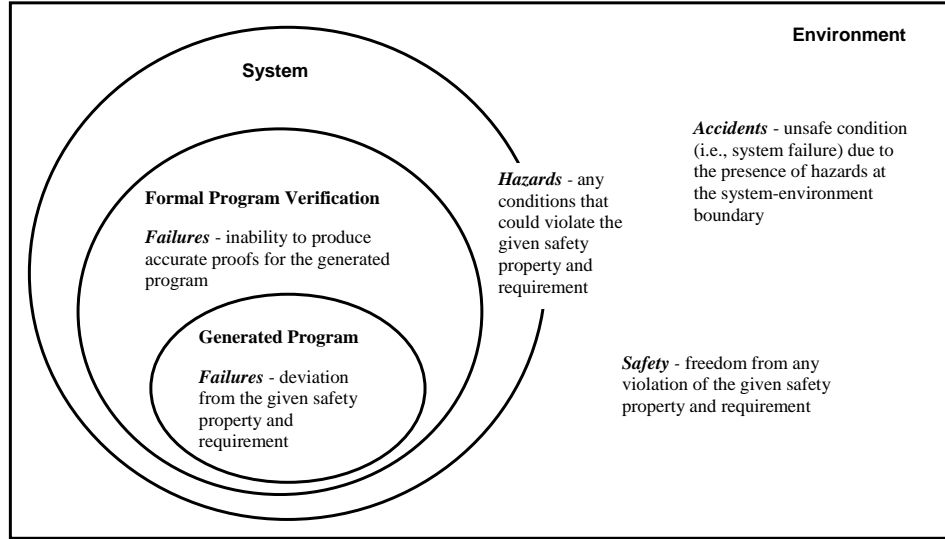


FIGURE 3.1: Position of Error, Fault, Failure, Hazard and Accident in the Formal Program Verification (adapted from [146])

system failure is defined as an accident and an undetected violation of the given safety requirement and property in the program as a hazard.

Accidents can occur in the environment due to the presence of hazards at the system-environment boundary. Any undetected violation of the given safety property or requirement (i.e., hazard) in the system that executes the program can lead to system failures and thus to the accident. A system or environment is called safe if it is free from any violation of the given safety property and requirement. Figure 3.1 illustrates the position of failures, hazards, accidents and safety [146] in the formal program verification context. It customizes the hazard analysis terminologies to suit our research context.

Since we rely on the certification system as a means to provide the assurance of the safety of the program, any conflicts, failures, and errors in the process of providing the proofs (e.g., missing unsafe location in the program, invalid axioms used in a proof, or a wrong property being proven) are undesired events that have the potential to cause the hazard. In our work, we define program failures as “deviation from the given safety properties and requirements” and certification failures as “inability to produce accurate proofs for the generated program”. An incorrect proof of an unsafe program as safe is the major failure of the certification system, as it fails to detect the program failures and to reject the unsafe program.

3.2 Hazard Analysis Methodology

In our work, we use a hazard analysis methodology [93] as shown in Figure 3.2 to guide us in the process of identifying and analyzing the program verification hazards. The

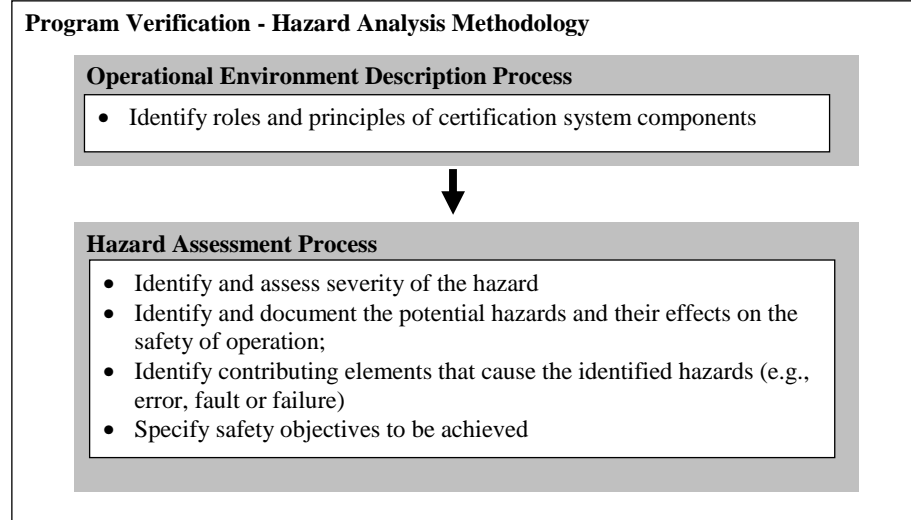


FIGURE 3.2: Hazard Analysis Methodology (adapted from [93])

methodology includes two main processes, namely the operational description process and the hazard assessment process. Figure 3.2 illustrates these processes and their corresponding activities. A detailed explanation of the processes is given in the next subsections.

3.2.1 Operational Description Process

The operational description process [93] identifies the roles and principles of the components in the formal software safety certification system. The objective of this process is to formulate an explicit understanding of the system's components and their functions, operational scenarios, and principal conditions. Based on the results of a system analysis, the certification system consists of 22 components.

These components can be categorized into five groups, based on their roles in the code generation and certification process (see Table 3.1). Table 3.1 also shows whether the components are tools or artifacts; the latter are either integral parts of the system, or derived by one of the tools in the system. First, the *code generation* category contains all components that support the generation of the program, such as the code generator that automatically generates and translates the code into a target platform. Second, the *annotation generation* category contains all components that are required for the construction of the annotations, in particular the annotation schemas and the annotation inference algorithm, which has been described in Chapter 2. Third, the *safety definition* category contains all components that are related to the definition of program safety. Fourth is the *VC generation* group of components that generate, simplify, and convert the VCs (for example that convert VCs into the particular first-order logic syntax of different ATPs), and finally, the *proof generation* components that are used to find and

TABLE 3.1: AFCG Component Groups

Role	Components	Tool/Artifact
Generate Code	Code Generator	Tool
	Symbolic Computation	Tool
	Schema	Artifact
	Code Fragment	Tool
	Intermediate Code	Artifact (derived)
	Optimization	Tool
	Generated Code	Artifact (derived)
Generated Annotation	Annotation Schema	Artifact
	Annotation Inference Algorithm	Tool
	Annotation	Artifact (derived)
Define Safety	Safety Property	Artifact
	Safety Requirement	Artifact
	Safety Policy	Artifact
	Safety Predicate	Artifact
Generate and Simplify VCs	VCG	Tool
	VC	Artifact (derived)
	Simplifier	Tool
	TPTP2X-Converter	Tool
Produce and Check Proofs	ATP	Tool
	Proof	Artifact (derived)
	Domain Theory	Artifact
	Proof Checker	Tool

validate the proofs of VCs, e.g., the ATP and proof checker. An explicit explanation of each component is provided in Appendix A. Note that the derived components capture the safety-relevant information about a specific program on different levels of abstraction (i.e., annotations, VCs, and proofs) but that the ultimate evidence of the safety cases we are constructing are the proofs.

3.2.2 Hazard Assessment Process

The hazard assessment process helps in identifying and assessing hazards and dealing with them [93]. The purposes of the process are:

- to identify and assess the severity of the hazards;
- to identify and document the potential hazards and their effects on the safety of operation;
- to identify contributing elements that cause the identified hazards (e.g., error, fault or failure);
- to specify the safety objectives to be achieved.

In our work, the hazard assessment process is divided into two parts, severity classification scheme and hazard analysis. The details of each part are described in the following sections.

3.3 Hazard Severity Classification Scheme for Program Verification

Categorizing hazards according to their severity level helps in a consistent assessment of hazard impacts and is also beneficial in identifying the level of improvement and corrective action that are required to control and mitigate the identified hazard [111, 132]. In this section, a dedicated severity scheme is constructed to help in assessing the severity level of the worst possible effect caused by the hazards identified in program verification. This scheme, called the *program verification hazard decision matrix* combines the perspective of users of the code generator and the perspective of users of the certification system as we have to consider the situation in both contexts. Our situation is complicated by the fact that the code generator is a meta-level system, and we do not know the application context of the generated program, and the fact that the working of the certification system also highly depends on the program to be verified. Moreover, the automatic certification is a complex process, which complicates an intuitive understanding of the assurance provided by the certification system. To understand the interaction between these two systems (i.e., code generator and certification system), we use the following set of indicators to assess the possible effects:

- the output of the code generator, in particular, whether the program actually *is* safe or unsafe; and
- the output of the certification system, in particular, its *claim* about the safety of the program as safe, unsafe or unknown.

In principle, the generator only produces two types of output, either a safe or an unsafe program and the certification system generates three types of output, i.e., a claim about the safety of the program either as safe, unsafe or unknown. In the certification system, an unknown claim is produced when it is unable to decide whether the program is safe or unsafe. This can happen because the information provided to the certification system is insufficient to conclude safety of the program and of course because the underlying first-order logic is undecidable.

All situations in which these two indicators (i.e., the output of code generator and the output of the certification system) do not agree are considered as abnormal, or as fault of the combined system (i.e., code generator and certification system). Therefore, careful judgement needs to be exercised in assessing the hazard impact on the system that

execute the program. Based on the situation here, there are three possible conflicts to be considered. The conflicts are:

- when the certification system claims all safety requirements and properties are proven but the program exhibits an unsafe behavior when executed; or
- when the certification system claims that one or more safety requirements or properties are violated but the program always exhibits a safe behavior when executed; or
- when the certification system is unable to provide any specific claims about the program at the end of the certification process.

The worst possible outcome is if the certification system fails to provide an accurate claim about the safety of an actually unsafe program, as the risk for the unsafe program to fail when executed or to lead to unintended behavior is extremely high.

The customized severity classification scheme helps us in defining the severity level caused by the disagreement of the two systems. Here we defined three severity levels based on the interaction of the two systems, and the assumption that the outputs of the systems can be characterized as “the program is safe and unsafe” and “the certification system claims that the program is safe or unsafe or unknown”, respectively. Level A denotes hazardous disagreement, level B benign disagreement, or indecision, and level C denotes agreement between both systems. The severity level of each situation or any conflict between these two systems is described in the program verification hazard decision matrix as shown in Figure 3.3:

- level C for a “safe” claim that is produced on a safe program;
- level A for a “safe” claim that is produced on an unsafe program;
- level B for an “unsafe” claim that is produced on a safe program;
- level C for an “unsafe” claim that is produced on an unsafe program; and
- level B for any “unknown” claim.

Based on the interaction severity level assessment, the most undesirable situation is when the certification system erroneously claims that the program is safe to be executed but it turns out that the program is unsafe when executed (i.e., severity level A). This might happen due to some faults or failures in the certification system such as incorrect proofs or incomplete certification coverage of the program. These unnoticeable faults can obviously undermine the assurance provided by the certification system. In contrast, the best situation (i.e., severity level C) is when the accurate judgment has been made

Program Behaviour	Safe	C	B	B
	Unsafe	A	C	B
		Safe	Unsafe	Unknown
		Certification System Claim		

FIGURE 3.3: Program Verification Hazard Decision Matrix

by the certification system for the program, both positive and negative, for example a claim that the program is safe to be executed, and the program is safe when executed. Such as accurate claims on safe or unsafe programs can thus establish confidence on the assurance provided by the certification system.

3.4 Fault Tree Analysis for Formal Program Verification

While formal program verification has become a viable alternative in demonstrating program safety, doubts about the trustworthiness of the verification proofs remain. These doubts concern not only the correctness of the proofs (i.e., whether each proof step is legal in the underlying calculus) or the correctness of any of the other tools that handle the verification conditions, but also the question whether the proofs actually entail program safety. Since there are many possible ways in which the trustworthiness can be compromised, a fault tree analysis is required to identify the chain of causes and their interaction logic that initiate this undesired event. The customized severity classification scheme defined in the previous section helps in the process of identifying and analyzing the hazards.

In order to analyze the situation at this meta-level (rather than deferring this to the final application), we need to make the simplifying but conservative assumption that every violation of a requirement (i.e., safety requirement and safety property) is a “potential condition that can cause harm to personnel, system, property or environment”, i.e., a hazard [94]. A further complication is caused by the fact that the certification system is purely observational in the sense that it cannot introduce any additional hazards as defined above, but should nonetheless be included in the hazard analysis. As outlined in the previous section, we thus need to look at the interaction between the code generator and the certification system to identify faults of the combined system. We consider all situations in which these two indicators do not agree as abnormal or faults of the combined system. The most critical fault (level A), on which we concentrate here, occurs if the code exhibits an unsafe behavior when it is executed but the certification system claims that all requirements (i.e., safety requirement and safety property) were proven to hold.

The fault tree shown in Figure 3.4 demonstrates the combinations of events that could lead to the top-level hazard (H1), (i.e., an undetected violation of the given requirements), are linked together. Here, the analysis focuses on showing possible events that might invalidate the safety claim construction as it follows the structure of the generated code.

Figure 3.4 shows that there are two potential causes for the top-level hazard, either the certification system missed a potentially unsafe location in the code (F1) or erroneously concluded that all locations in the code are safe (F2). Potentially unsafe locations can be missed because of:

- E1: an incomplete or incorrect formalization and localization of the safety requirement (e.g., an incorrect formula); or
- E2: an incomplete coverage of the program, which can be caused by missing claims for any variable (E2.1), variable occurrence (E2.2), or path (E2.3) in the program; or
- E3: missing VCs (e.g., due to errors in the VCG); or
- E4: an incomplete or incorrect representation of the safety requirements in critical annotations (e.g., a wrong global post-condition on the output variables); or
- E5: an incomplete or incorrect formalization of the safety policy corresponding to the given safety property (e.g., the failure to detect a location as potentially unsafe).

Since any location is considered safe if a proof for its corresponding safety obligation can be found, assuming the hypotheses available at that location, the conclusion that the program is safe at all locations can be wrong due to four reasons:

- E1: an incomplete requirement - note that this same hazard can also cause F1; or
- E6: the hypotheses used in the proof can be wrong (i.e., do not hold at the location); or
- E7: the proof can be technically wrong (i.e., does not conform to the inference rules of the underlying calculus); or
- E8: the safety obligation that is proven can be wrong (i.e., does not imply the safety of the location).

The hypotheses can be wrong because of:

- E6.1: incorrect hypotheses from definition because of:

- E6.1.1: the hypotheses result from a wrong definition location i.e., a definition is not connected to the calculated use location by the correct path; or
- E6.1.2: the hypotheses have been constructed wrongly at a definition; or
- E6.2: the hypotheses are not maintained along the paths from the definition to the calculated use; or
- E6.3: the different hypotheses from the different paths are inconsistent to each other.

A proof can be incorrect due to the use of:

- E7.1: an invalid domain theory which can be caused by:
 - E7.1.1: incorrect specification of library functions; or
 - E7.1.2: invalid axioms for proving the proofs; or
 - E7.1.3: invalid certification assumptions in deriving the proofs; or
- E7.2: an incorrect mechanism which can be caused by:
 - E7.2.1: unreliable theorem prover (i.e., produce invalid proofs); and
 - E7.2.2: unreliable proof checker to detect invalid proofs).

Note that we subsume all auxiliary infrastructure for the ATP (e.g., the TPTP2X converter) under E7.2.1 as well.

The safety obligation can be wrong if:

- E8.1: any of the critical annotations are wrong (similar to the case of missing a potentially unsafe location described above); or
- E8.2: the formalization of safety policy is inadequate; or
- E8.3: the safety predicate is wrong; or
- E8.4: its implementation in the VCG are wrong.

E8.4 also subsumes other errors in the implementation of the VCG, e.g., its formula handling. The completeness of the fault tree can be checked by a bottom-up analysis, i.e., Failure Mode Effect Analysis (FMEA) [94, 132, 143]. FMEA analyzes the effects of each single failure on component and the seriousness of this failure mode to the overall system. However, we defer the implementation of bottom-up analysis for future work.

3.5 Conclusions

Formal methods such as formal software safety certification [43, 46, 47] can be used to demonstrate safety of the generated code in the sense that the execution of the code does not violate a specified safety property or safety requirement, by providing formal proofs as explicit evidence or certificates for the assurance claims. However, in practice there are reservations about the use of formal proofs as evidence (or even arguments) in safety-critical applications. Concerns that the proofs may be based on assumptions that are not valid, or may contain steps that are not justified, can undermine the reasoning used to support the assurance claim and complicate an intuitive understanding of the assurance claims provided by the proofs. Moreover, the complexity of the tools used can lead to unforeseen interactions and thus causes additional concerns about the trustworthiness of the assurance claims. In this chapter, we have identified possible events that might invalidate the safety claim construction. We used the hazard analysis technique, in particular the fault tree analysis to identify possible faults and failures to the program safety and the certification process, as well as their interaction logic that initiate these undesired events. The level of details of each undesired event identified in the fault tree can be expanded further, however, we left this for future work. In the following chapters, we construct safety cases that show all the undesired events have been considered, controlled and perhaps mitigated in the formal program verification phase. This can increase trust in using the formal program verification method to provide assurance for the safety of the program.

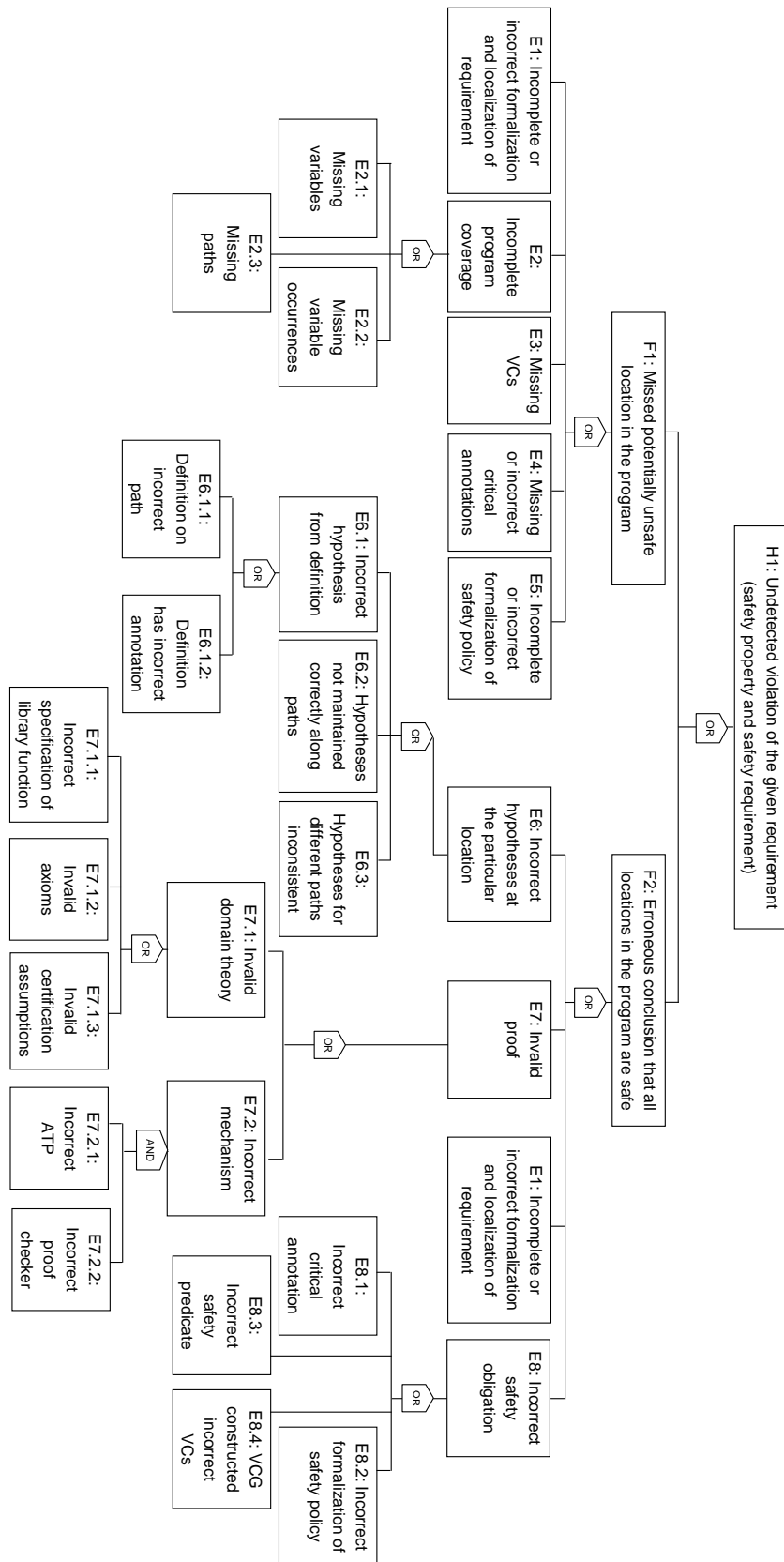


FIGURE 3.4: Fault Tree for Program Verification

Chapter 4

Property-Oriented Safety Cases

Chapter 4 presents an approach to systematically construct safety cases from information collected during a formal safety certification of code, in particular from the construction of the logical annotations necessary for a Hoare-style verification. Here, we describe the structure of the safety cases that focus on demonstrating safety of the generated code wrt. a given safety property. This chapter also presents the argument patterns for the property-oriented safety cases.

4.1 Introduction

Formal methods such as formal program verification can in principle provide the highest levels of assurance of code safety by providing formal proofs as explicit evidence for the assurance claims. However, several problems remain, as described in Chapter 1, in particular for the use of program verification in auto-generated code. Here, we address these problems and present an approach to systematically derive safety cases from information collected during the formal program verification phase, in particular the construction of the necessary logical annotations. In this chapter, our focus is on showing safety of the program wrt. a given safety property. In Chapter 3, we have used a fault tree analysis to identify possible risks to the program safety and the certification process, as well as their interaction logic. We use the results of this analysis to guide the construction of the safety cases, but their precise structure is derived from the construction of the logical annotations, and reflects the way in which the annotation inference has analyzed the program. We instantiate our generic argument structure for code generated by AutoFilter system [153] developed at NASA Ames. We then generalize this approach and describe safety case patterns for the property-oriented safety cases.

In this chapter, we use the initialization-before-use safety property as running example, but our framework can handle a variety of other safety properties, including absence

of out-of-bounds array accesses [43]; we expect that other properties handled by proof-carrying code such as null pointer dereferences [106] can be formalized easily. However, we are not restricted to showing exception freedom but can also encode domain-specific properties such as matrix symmetry, measurement unit or coordinate frame consistency (which requires significant proofs involving matrix algebra and functional correctness), whose violation will not immediately cause a run-time exception but still renders the code unsafe. These properties are called domain specific properties because they only apply in the domain of the code generator. For example, matrix symmetry applies to the state estimation domain of the AutoFilter but not to the Guidance, Navigation, and Control (GN&C) systems generated by Real-Time Workshop discussed in Chapter 5 and 6. The only assumption from the underlying certification framework that we use is that the safety of the program can be expressed in terms of conditions on its variables.

AutoFilter: Synthesis of State Estimation Software. AutoFilter system is a domain-specific synthesis system that is implemented in SWI-Prolog [154]. It takes high level specifications of state estimation problems and derives code by repeating application of schemas i.e., algorithms and solution methods of the domain [153]. The schemas return the code fragment in AutoFilter’s intermediate language. The code fragments are formulated in an intermediate language that is essentially a “sanitized” variant of C (e.g., neither pointers nor side-effects in expressions) but also contains a number of higher-level domain-specific constructs (e.g., vector or matrix operations, finite sums, and convergence loops) [50]. The code fragments resulting from the applications of the individual schemas are assembled and the synthesizer will automatically generate and translate code into the targeted platform, for instance C and Modula-2. The resulting code uses variants of the Kalman filter algorithm [31] that are appropriate for the specified state estimation problem and typically comprises approximately 500-1500 lines of code including auto-generated comments [50].

To support the automatic program verification, AutoFilter is extended with the standard Hoare-style program verification components as described in Section 2.4.2. The formal program verification works on the source code level to demonstrate that a program meets the given safety property and remains safe during its execution. Figure 4.1 shows the example of a code fragment and annotation generated by AutoFilter system from a simplified model of the Crew Exploration Vehicle dynamics [42].

Initialization-Before-Use Safety. In this chapter, we show how to construct safety cases for the initialization-before-use safety property, i.e., we check whether each variable or individual array element has explicitly been assigned a value before it is used. In general, in order to ensure a program is safe wrt. a safety property, the annotations must formalize all pertinent information that is necessary for the ATP to prove that all *potentially* unsafe locations are in fact safe. This information will be established at some definition location in the program and maintained along all control-flow paths to all the potentially unsafe locations, where it is used. For initialization-before-use,


```

...
5 const M=6, N=12;
...
<init h>
183 post  $\forall 0 \leq i < M, 0 \leq j < N \cdot h_{\text{init}}[i, j] = \text{INIT}$ 
...
<init r>
525 post  $\forall 0 \leq i, j < M \cdot r_{\text{init}}[i, j] = \text{INIT}$ 
...
683 while t < Tmax
    inv  $\forall 0 \leq i < M, 0 \leq j < N \cdot h_{\text{init}}[i, j] = \text{INIT} \wedge \dots \wedge \forall 0 \leq i, j < M \cdot r_{\text{init}}[i, j] = \text{INIT} \wedge \dots$  do
    ...
728 for k:=0 to N-1
    inv  $\forall 0 \leq i < M, 0 \leq j < N \cdot h_{\text{init}}[i, j] = \text{INIT} \wedge \dots \wedge \forall 0 \leq i, j < M \cdot r_{\text{init}}[i, j] = \text{INIT} \wedge \dots$ 
     $\wedge \forall 0 \leq i, j < N \cdot i < k \Rightarrow u_{\text{init}}[i, j] = \text{INIT} \wedge d_{\text{init}}[i, j] = \text{INIT}$  do
729 for l:=0 to N-1
    inv  $\forall 0 \leq i < M, 0 \leq j < N \cdot h_{\text{init}}[i, j] = \text{INIT} \wedge \dots \wedge \forall 0 \leq i, j < M \cdot r_{\text{init}}[i, j] = \text{INIT} \wedge \dots$ 
     $\wedge \forall 0 \leq i, j < N \cdot (i < k \vee i = k \wedge j < l) \Rightarrow u_{\text{init}}[i, j] = \text{INIT} \wedge d_{\text{init}}[i, j] = \text{INIT}$  do
730 u[k, l] := 0;
731 d[k, l] := 0;
    post  $\forall 0 \leq i < M, 0 \leq j < N \cdot h_{\text{init}}[i, j] = \text{INIT} \wedge \dots \wedge \forall 0 \leq i, j < M \cdot r_{\text{init}}[i, j] = \text{INIT} \wedge \dots$ 
     $\wedge \forall 0 \leq i, j < N \cdot i \leq k \Rightarrow u_{\text{init}}[i, j] = \text{INIT} \wedge d_{\text{init}}[i, j] = \text{INIT}$ 
    post  $\forall 0 \leq i < M, 0 \leq j < N \cdot h_{\text{init}}[i, j] = \text{INIT} \wedge \dots \wedge \forall 0 \leq i, j < M \cdot r_{\text{init}}[i, j] = \text{INIT} \wedge \dots$ 
     $\wedge \forall 0 \leq i, j < N \cdot u_{\text{init}}[i, j] = \text{INIT} \wedge d_{\text{init}}[i, j] = \text{INIT}$ 
    ...
    <use u, d>
    ...
    <use h, ..., r>
    ...
end;
```

FIGURE 4.1: Code Fragment and Generated Annotation [50]

definitions correspond to variable initializations while uses are statements which read a variable. The annotations use shadow variables to record the initialization status of the program variables, so that for example $X_{\text{init}} = \text{INIT}$ records the fact that X is (or must be) initialized. An invariant for a for-loop might express that an array has been initialized up to the loop index as $\forall j \leq i \cdot A_{\text{init}}[j] = \text{INIT}$.

The program [42] shown in Figure 4.1 initializes some of the vectors and matrices (such as **h** and **r** cf. lines 5-525) with model-specific values before they are used and potentially updated in the main while-loop. It also uses two additional matrices **u** and **d** that are repeatedly zeroed out and then partially recomputed before they are used in each iteration of the main loop (lines 728-731). For initialization safety, the annotations need to formalize that each of the vectors and matrices is fully initialized after the respective code blocks. For the loops initializing **u** and **d**, invariants formalizing their partial initialization are required to prove that the postcondition holds.

The VCG will turn these annotations into VCs corresponding to establishing the invariant on loop entry, preservation of the invariant by the loop body, and implication by the “exit form” of the invariant (i.e., over the loop bounds) of the loop post-condition. However, it is the dependencies between the definitions and the use locations, rather than the annotations or the VCs, which govern the overall structure of both the safety argument and the safety case.

4.2 Constructing Safety Cases for Demonstrating a Safety Property

In our work, we consider each violation of the given safety property (e.g., use of an uninitialized variable) by the generated code as a hazard. The purpose of the safety cases here is to show that the safety property is in fact not violated and thus that the risk associated with this hazard (as identified in Section 3.4) is controlled or mitigated and can not lead to a system failure. The safety case makes explicit the formal and informal reasoning principles, and reveals the top-level assumptions and external dependencies that must be taken into account. It also provides information about why the generated code can be assumed to be sufficiently safe. The structure of this argument is constructed from information collected by the annotation inference algorithm. However, the evidence still comes from the formal safety proofs.

We build a generic, multi-tiered argument and instantiate it with respect to a given safety property and program. Its three tiers together constitute a single safety case that justifies the safety of the program. The upper tier simply instantiates the notion of safety and the formal definitions for the given safety property; in particular, it explicitly describes the context and constraints of the Hoare-style partial correctness proofs. The two lower tiers argue the safety of the program as governed by the property. The lower tiers are constructed individually to reflect the program structure (i.e., the variables and their relevant occurrences). This can be done systematically because their structure directly follows the course the annotation construction takes through the program.

4.2.1 Tier I: Explaining the Safety Notion

Figure 4.2 shows the the top tier of the safety case. It starts with the top-level safety goal (i.e., the safety of the generated code with respect to the safety property of interest) and shows how this is achieved by a defensible argument based on the partial correctness of the generated code. The argument stresses the meaning of the Hoare-style framework, specialized to the given safety property. However, the argument structure remains independent of the property. Here, contexts explain the informal interpretation of key notions like “safe” and “safety property”. Constraints outline limitations of the approach, in particular, the fact that certification works on an intermediate representation of the source code and only shows a single property, e.g., initialization-before-use. Hyperlinks can refer to additional evidence in the form of documents containing, for example, the model from which the source code has been generated as an informal description of the safety property.

The key strategy at this tier and its model (i.e., a Hoare-style partial correctness proof using the dedicated proof rules of the init-before-use safety policy) as well as its limi-

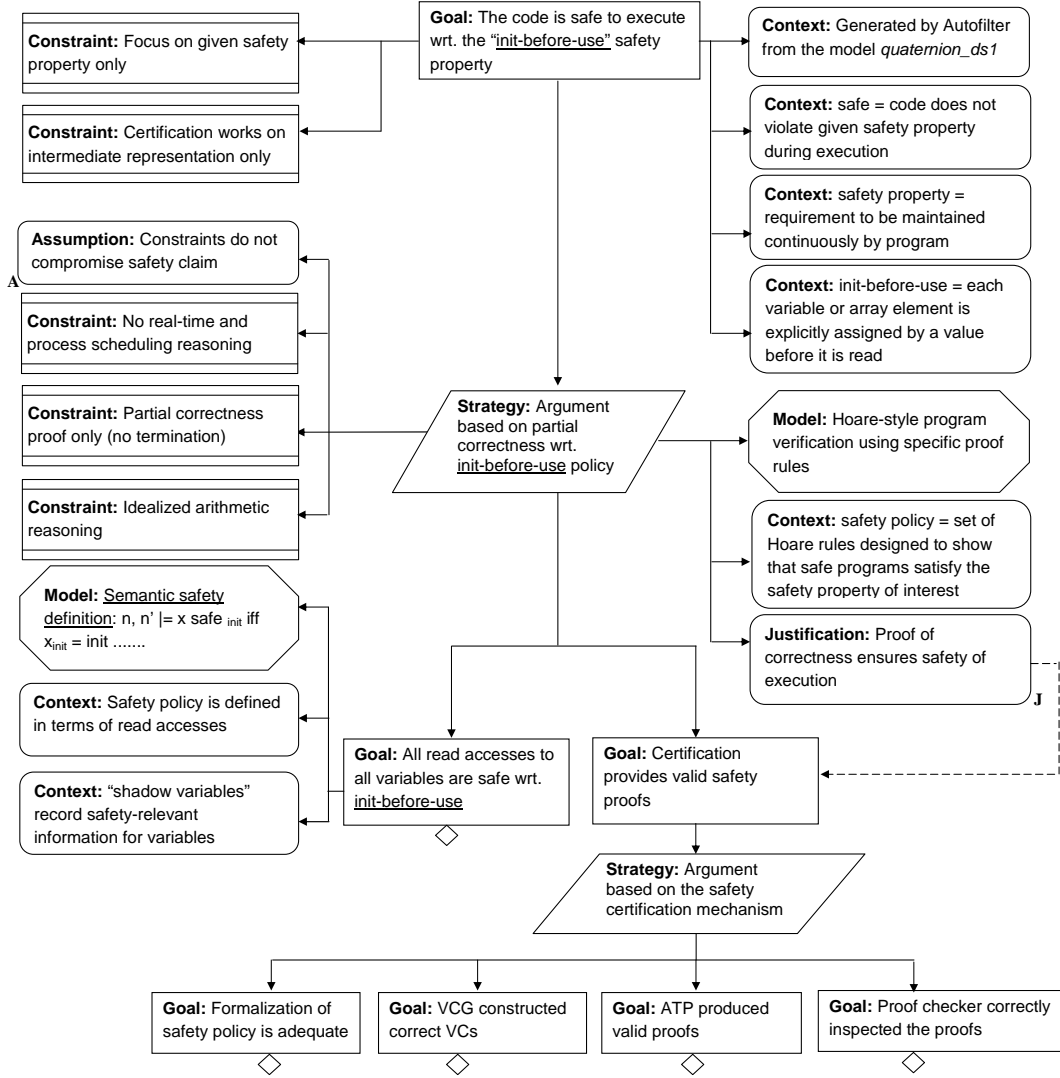


FIGURE 4.2: Tier I of Derived Safety Case: Explaining the Safety Notion

tations (i.e., no termination proof, no real-time and process scheduling reasoning and consider only idealized arithmetic reasoning) and assumption (i.e., constraints do not compromise safety claim) are made explicit. The strategy reduces showing the safety of the whole program to showing the safety of all read accesses, which emerges as first subgoal. This is based on the fact that the safety property is defined in terms of variable read accesses which is given as a context to the goal. The subgoal is further elaborated by a model of the semantic safety definition, which exactly defines what is meant by “safe”, using the notion of shadow variables given as context. In Figure 4.2, this is in abbreviated form; a hyperlink gives access to the definition and explanation, as for example in [43]. The strategy’s second subgoal is to show that the certification system provides valid proofs of the program safety, which is also the foundation of the strategy’s original justification (i.e., the claim that the proofs ensure the safe execution of the program). This goal is further decomposed into argument based on the mechanism used for verifying the program, which thus leads to four subgoals, i.e., that the safety policy

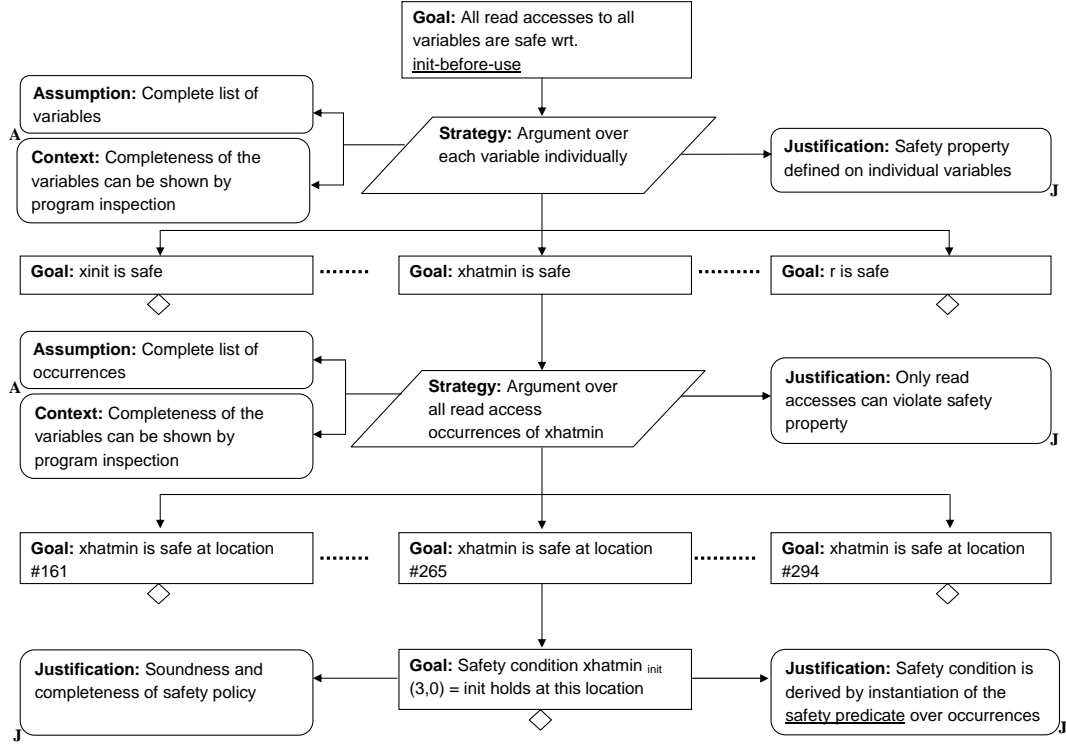


FIGURE 4.3: Tier II of Derived Safety Case: Arguing over the Variables

adequately represents the safety property, the VCG constructed correct VCs, the ATP produced valid proofs of the VCs and the proof checker correctly inspected the validity of the proofs. The further argument structure of these subgoals justify the claim that the undesired events E5 and E8.2, E8.4, E7.2.1 and E7.2.2 identified in the fault tree analysis (cf. Figure 3.4, page 53) have been controlled in the formal program verification process. However, these subgoals are not elaborated further in this safety case but lead to a complementary safety case for the safety logic and safety proofs. The argument structure for the adequacy of the safety policy could follow the lines of [43]. The second leg of the argument can be considered as backing evidence to the first subgoal and could be represented as an away goal. We choose to represent the argument as shown in Figure 4.2 to clearly point out the role of the deferred arguments in this leg.

4.2.2 Tier II: Safety of Program Variables

The second tier reduces the safety of all variables in two steps, first to the safety of each individual variable (justified by the fact that the safety property is defined on individual variables) and then to the safety of the individual occurrences. Note that the number of subgoals of both strategies (see Figure 4.3 for the goal structure) and the safety conditions are program-specific. This information is provided by the annotation inference.

Both strategies are predicated on the assumption that they iterate over the complete list

of variables resp. occurrences. Each individual occurrence then leads to a subgoal to show that the computed safety condition is valid at the location of the variable's occurrence. This reduction to a formal proof obligation is justified by the soundness and completeness of the safety policy; in addition, the specific form of the safety condition is also justified. Note that some of the root causes identified in the fault tree remain as assumptions in the safety case (i.e., the list of variables and their occurrences are assumed to be complete). However, these can be checked easily, e.g., by a simple program inspection, since they require no deep analysis of the generated code; in fact, the check could be automated easily. The safety case records this as a context to the strategy. Therefore, the argument structure shown mitigates any doubts on the incomplete program verification coverage (as shown in Figure 3.4, page 53).

4.2.3 Tier III: Sufficiency of Safety Condition

The final tier (see Figure 4.4 for the goal structure) argues the safety of each individual variable access, using a strategy based on establishing and maintaining appropriate annotations. The purpose of the argument is to show that the undesirable events that initiate E8 (primarily), and E4 and E6 (cf. Figure 3.4), have been controlled and mitigated. The argument structure directly reflects the course the annotation inference has taken through the code. The first subgoal is thus to show that the variable safety is established on all paths leading to the current location, using an argument over all definition locations (the argument of this subgoal justifies that the undesired events E6.1, E6.1.1 and E6.1.2 (as shown in Figure 3.4, page 53) have been controlled). Here, the model for the subgoal corresponds to the annotation schema that was applied during annotation inference to identify the definition. Each definition thus leads to a corresponding subgoal and then further to any number of VCs based on the strategy of the generation of the VCs corresponding to the initialization location in the program, although here only a single VC emerges in both cases. The proofs from these VCs demonstrate that the top event identified in the hazard analysis does not occur for the given program.

The second subgoal of the top-level strategy here is to show that the established variable safety is maintained along all paths. The argument of this subgoal justifies that the undesired event of E6.2 (as shown in Figure 3.4, page 53) has been controlled. This proceeds accordingly and the proofs of the VCs again demonstrate that the identified top event is mitigated. The final subgoal is then to show that the variable safety implies the validity of the safety condition. The argument of this subgoal justifies the undesired events of E6.3 (as shown in Figure 3.4, page 53) has been controlled. This can again lead to any number of VCs.

Note how goals that concern properties of the program (e.g., “xhatmin is safe”) are decomposed into subgoals that comprise program-independent tasks for the prover, i.e., VCs. The validity of the construction of the VCs depends on the soundness of the rules of

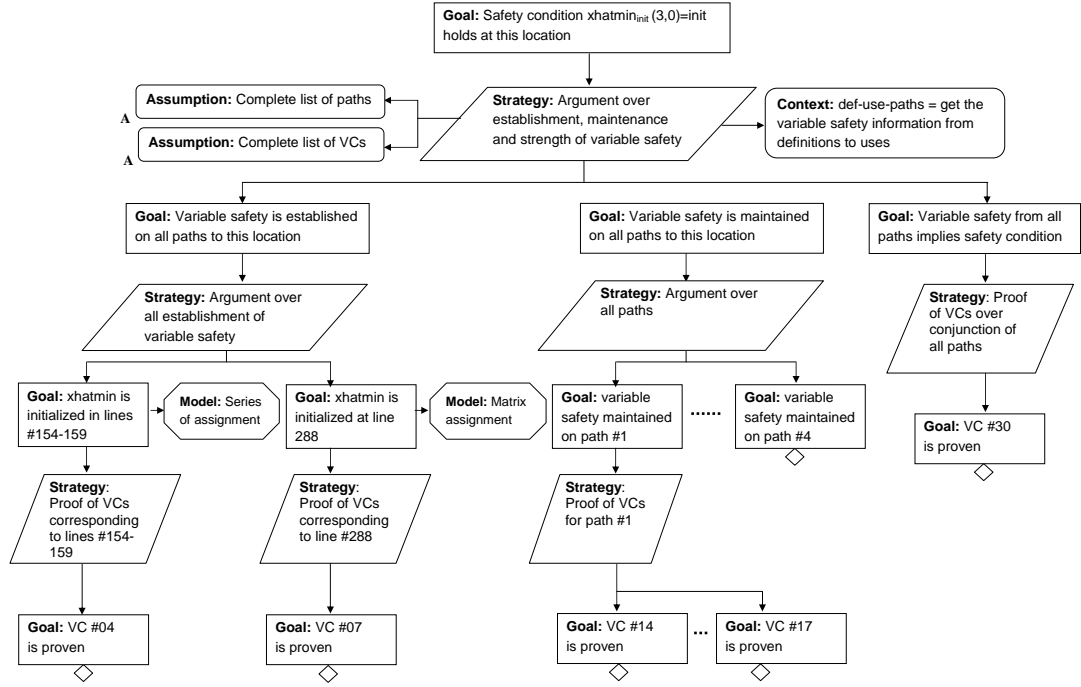


FIGURE 4.4: Tier III of Derived Safety Case: Arguing over the Paths

the VCG, the simplifier, and the definition of the safety policy, while the correspondence to program locations is based on tracing information added by the VCG and retained during the certification process. If (and only if) all VCs can be shown to hold, then the safety property holds for the entire program. The evidence for the safety case is provided by the formal proofs of the VCs.

4.3 Safety Case Patterns for Property-Oriented Safety Cases

Safety case patterns generalize the structure of the property-oriented arguments i.e., for arguing safe execution of the generated code with respect to the given safety property. Here patterns are used as a way of abstracting the fundamental strategies from the details of the argument framework. Patterns are based upon reusable goal structures that can be instantiated to aid the construction of parts of an argument [86, 88, 146]. Some of nodes in the property-oriented arguments are fixed, i.e., can be reused in new contexts or applications without being changed much from the original but some of nodes need to be further instantiated and developed. The patterns are generic and customizable as the general safety considerations at each tier remain unchanged.

4.3.1 Safety Case Pattern: Explaining the Safety Notion

This pattern generalizes the safety argument for explaining the safety notion of the program (as shown in the example in Figure 4.2). The main focus of the argument is to describe the approach that was used in ensuring the safety of the program. It stresses the meaning of the verification approach (i.e., Hoare-style framework), the contexts and constraints of the approach. The structure of the arguments remain the same as shown in the example Figure 4.2, but we generalize the contents of the safety case nodes. The argument pattern can be customized for other programs and safety properties by changing certain information such as the name of the safety property (i.e., $\langle \text{property-x} \rangle$) and the program (i.e., $\langle \text{module-x} \rangle$) in the pattern. However, while the contents of some pattern nodes are fixed (i.e., can be reused in new contexts or applications) but they need to be further instantiated (such as labelled with triangle \triangle) or further developed (such as labelled with diamond \diamond). The link with \circ describes the node that is optional as it depends on the safety property. Tables 4.1 to 4.5 describe in details the safety case pattern for explaining the safety notion.

4.3.2 Safety Case Pattern: Safety of Program Variables

This pattern generalizes the safety argument for arguing that the program safety can be reduced to the safety of all program variables. The structure of the arguments remain the same as shown in the example Figure 4.3, but we generalize the contents of the safety case nodes. The argument pattern can be customized for other programs and safety properties by changing certain information such as the definition of safety wrt. the given safety property (i.e., $\langle \text{property-x} \rangle$), the program (i.e., $\langle \text{module-x} \rangle$), the name of the variable (i.e., $\langle \text{var-x} \rangle$), the location of the variable in the program (i.e., $\langle \text{loc-x} \rangle$) and the established safety condition (i.e., $\langle \text{safety-condition-x} \rangle$). However, the contents of some pattern nodes are fixed (i.e., can be reused in new contexts or applications) but some of them need to be further instantiated (labelled with triangle \triangle) and further developed (labelled with diamond \diamond). The link with \circ describes the node that is optional as it depends on a given safety property. Typically, semantic safety definition is established for the language specific properties to give an explicit meaning of the properties. While the link with \bullet describes the node that can be solved by zero or more instances as the number of subgoals and strategies are program-specific and reflect the program structure. The safety argument here can be constructed systematically from information collected during the formal verification of the code. Tables 4.6 to 4.8 describe in details the safety case pattern for arguing safety of program variables.

4.3.3 Safety Case Pattern: Sufficiency of Safety Condition

This pattern generalizes the safety argument for that the safety conditions hold. The overall structure of the arguments remain the same as shown in the example Figure 4.4, but we generalize the contents of the safety case nodes. The argument pattern can be customized for other programs, safety properties and safety conditions by changing certain information such as the established safety condition (i.e., $\langle \text{safty-condition-x} \rangle$), the line of code (i.e., $\langle \text{loc-x} \rangle$), the variable (i.e., $\langle \text{var-x} \rangle$), the definition location (i.e., $\langle \text{defloc-x} \rangle$), the annotation schema that was applied during annotation inference to identify the definition (i.e., $\langle \text{annotation-schema} \rangle$) and the VCs constructed for the corresponding definition, maintenance and implication of variable safety (i.e., $\langle \text{vc-x} \rangle$). The generalization of the structure follows the same principle as described in Section 4.3.2. Tables 4.9 to 4.11 describe in details the safety case pattern for arguing sufficiency of safety condition.

4.4 Conclusions

We believe that formal methods such as formal software safety certification can provide the highest level of assurance of the code's safety. However, neither the methods by themselves nor the formal proofs they entail are a panacea, and it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claim and the proof rest. Therefore, in this chapter, we have described an approach to construct safety cases demonstrating the code's compliance with a specified safety property, using information collected during the automated construction of the logical annotations necessary for a Hoare-style verification. The safety case argues both the safety of the applied framework and the program. For the latter, the underlying argumentation structure is derived from the course the annotation inference has taken through the code. We also have constructed safety case patterns that are generic wrt. a given safety property. Each pattern addresses a decomposition step of the safety argument for the property-oriented safety cases.

In our work, we show that assurance is not implied by the trust in the generator but follows from an explicitly constructed argument for the generated code. We illustrate our approach on code generated by AutoFilter system [153] and use initialization-before-use safety as a running example, but our approach in principle is independent of the underlying generator and program. The same technique can be applied to other generators as the underlying annotation inference algorithm has also been applied to code generated from Simulink models using Real-Time Workshop. However, safety of program wrt. a given safety property only is insufficient, as some requirements cannot easily be expressed as safety property. Therefore, in the next chapter, we extend our work to construct safety cases from a given safety requirement.

TABLE 4.1: Safety Case Pattern: Explaining the Safety Notion

Explaining the Safety Notion			
Author	Nurlida Basir with the help of Ewen Denney and Bernd Fischer		
Created	05/11/2007	Last Modified	05/04/2010
Intent	The intent of this pattern is to provide a top level decomposition of the safety case that argues the program is safe wrt. a given safety property. The focus is to explain the context of the Hoare-style framework and notion of safety.		
Also Known As			
Motivation	<p>The motivation for this pattern is the need to show that the program is safe wrt. a given safety property by providing:</p> <ul style="list-style-type: none"> • formal proofs of the safety of the program wrt. a given safety property, and • assurance of the correctness of the certification mechanism used in providing the proofs. 		

TABLE 4.2: Safety Case Pattern: Explaining the Safety Notion

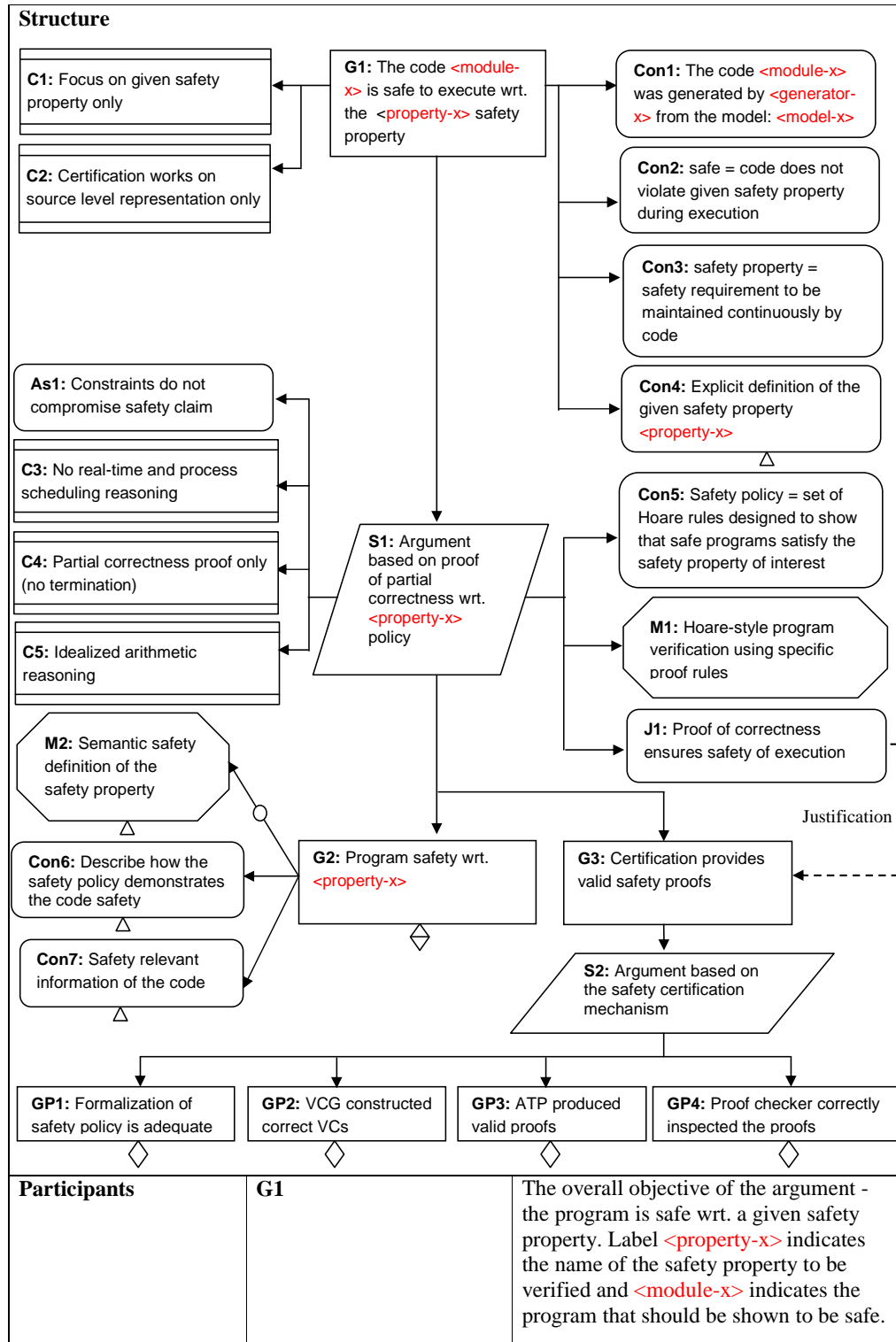


TABLE 4.3: Safety Case Pattern: Explaining the Safety Notion

G2	The objective of this claim is to show that the program is safe to execute wrt. a given safety property <property-x> by showing that all possible unsafe locations in the program are in fact safe and provides formal proof as evidence.
G3	This claim asserts that the Hoare-style program verification provides valid safety proofs for the program.
GP1	This claim asserts that the formalization of the safety policy is adequate and thus validates the approach adopted.
GP2	This claim asserts that the VCG constructed correct VCs.
GP3	This claim asserts that the automated theorem prover produced valid proofs of the VCs.
GP4	This claim asserts that the proof checker correctly inspected the proofs. In principle, it asserts that the proof checker identified all invalid proofs.
S1	Presents the strategy (based on proof of partial correctness) adopted to support G1 .
S2	Present the strategy for arguing the correctness of the certification mechanism used in providing the safety proofs.
As1	Assumes that all constraints do not compromise safety claim provided.
Con1	Describes information related to the program, the generator (the name of the generator) and the model used to generate the program.
Con2	Defines the notion of “safe” in relation to the program verification.
Con3	Defines the notion of “safety property” in relation to the program verification.
Con4	Provides the explicit definition of the given safety property <property-x> .
Con5	Defines the notion of “safety policy” in relation to the program verification.
Con6	Describes the relation of the safety policy to the given safety property, in term of how the safety policy demonstrates the code safety.

TABLE 4.4: Safety Case Pattern: Explaining the Safety Notion

	Con7	Describes the safety relevant information of the software e.g., about shadow variable.
	C1	Outlines constraint of the argument that the certification focuses on the given safety property only.
	C2	Outlines constraint of the argument that the certification works on source level representation only.
	C3	Outlines constraint of the argument that certification does not provides any real-time and process scheduling reasoning.
	C4	Outlines constraint of the argument that it provided partial correctness proof only, without requiring the algorithm to terminate.
	C5	Outlines constraint of the argument that certification presented an idealized arithmetic reasoning only.
	J1	Justification for the strategy S1 that the proof of correctness ensures safety of execution. This justification can be validated by G3 .
	M1	Describes the technique used for program verification, i.e., Hoare-style program verification using specific proof rules.
	M2 (optional)	Describes the semantic safety definition of the safety property.
Collaborations	<ul style="list-style-type: none"> • Con1, Con3 and Con4 describe the definition that supports the basis argument of G1. • C1, C2, C3, C4 and C5 impose restrictions to which extend G1 can be achieved. • As1 is statement that has to be relied upon in order to make the argument S1 that shows G1 valid. 	
Applicability	This pattern is applicable for describing the notion of safety of the program and outlines the context, constraints and assumptions of the formal program verification approach in showing program safety.	
Consequences	<p>After instantiating this pattern a number of undeveloped goals will remain:</p> <ul style="list-style-type: none"> • G2 In accordance with the top level claim of the pattern, this goal must be developed to give a complete safety argument for the safety of the code. • GP1-GP4 To support the decomposition of the top level goal (G1) into sub-goal (G2), it is necessary to support the decomposition of G3. The decomposition of G3 can assure that the certification provides valid safety proofs. 	

TABLE 4.5: Safety Case Pattern: Explaining the Safety Notion

Implementation	<p>This pattern should be instantiated in a top down approach. Most of the texts in this pattern are boiler-plate as the safety considerations at this level are same for all programs. However some texts (highlighted in red) such as Con4, M2, Con6 and Con7 need to be further instantiated wrt. a given safety property.</p> <p>Possible Pitfalls</p> <ul style="list-style-type: none"> • Attempting to decompose G1 in sub-goals over G2 without adequately showing GP1.
Examples	As shown in Section 4.2, Figure 4.2
Known Uses	
Related Patterns	<p><i>Safety of Program Variables</i> - This pattern can be used to decompose the undeveloped goal G2</p> <p>This pattern forms part of a program certification of auto-generated code pattern catalogue, which includes the following patterns:</p> <ul style="list-style-type: none"> ▪ Safety of Program Variables ▪ Sufficiency of Safety Condition

TABLE 4.6: Safety Case Pattern: Safety of Program Variables

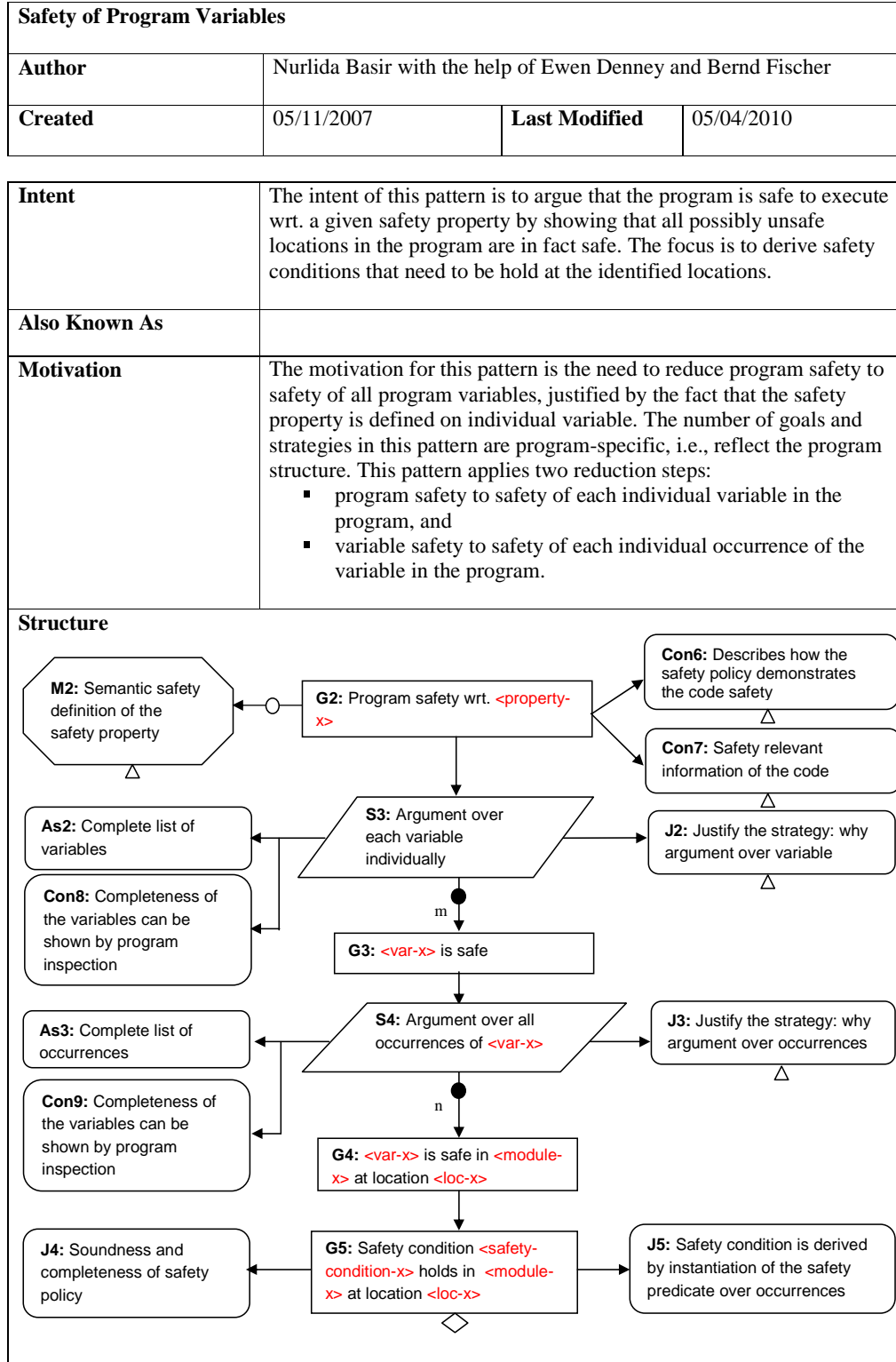


TABLE 4.7: Safety Case Pattern: Safety of Program Variables

Participants	G2	The objective of this claim is to show that the program is safe to execute wrt. a given safety property <property-x> , i.e., that there are no violations of the safety property.
	G3 (m of)	This claim asserts that the variable <var-x> is safe. The number of goals is program-specific i.e., reflects to the number of relevant variables in the program <module-x> .
	G4 (n of)	This claim asserts that the variable <var-x> is safe at location <loc-x> in module or code <module-x> . The number of goals is program-specific i.e., reflects to the number of relevant occurrences of the variable in the program.
	G5	This claim asserts that the safety condition <safety-condition-x> holds for module or code <module-x> at location <loc-x> of code.
	S3	Decomposes the program safety into the safety of each individual relevant variable in the program.
	S4	Decomposes the variable safety into the safety of all relevant occurrences of variable.
	As2	Assumes the argument iterates over the complete list of relevant variables.
	As3	Assumes the argument iterates over the complete list of relevant occurrences.
	J2	Justifies the strategy: how the program variable can demonstrate that the program is safe wrt. a given safety property.
	J3	Justifies the strategy: how the program variable occurrences demonstrate that the software is safe wrt. a given safety property.
	J4	Describes that the safety policy is sound and complete.
	J5	Describes that the safety condition is derived by instantiation of the safety predicate over occurrences.
	Con6	Describes the relation of the safety policy to the given safety property, in term of how the safety policy demonstrates the code safety.
	Con7	Describes the safety relevant information of the program.
	Con8, Con9	Describes that the complete list of variables and occurrences can be checked by program inspection.
	M2 (optional)	Describes the semantic safety definition of the safety property. The existence of M2 in the argument depends on the safety property.

TABLE 4.8: Safety Case Pattern: Safety of Program Variables

Collaborations	<ul style="list-style-type: none"> • M2, Con6 and Con7 are necessary to support the implication of G2. • As2 is a statement that has to be relied upon in order to make the argument S3 that shows G2 valid. • As3 is a statement that has to be relied upon in order to make the argument S4 that shows G3 valid. • J2 justifies the rationale of the strategy adopted to support G2. • Con8 and Con9 justify the strategy S3 and S4 respectively, i.e., that complete list of variables and occurrences can be checked by program inspection. • J3 justifies the rationale of the strategy adopted to support G3. • J4 and J5 justify that G5 is valid. • J5 describes how safety condition in G5 is instantiated.
Applicability	This pattern is applicable for arguing program safety to safety of all program variables, justified by the fact that the safety property is defined on individual variables. It is not applicable for safety properties that cannot be expressed in terms of the safety of variables, or cannot be reduced to the safety of individual variables.
Consequences	<p>After instantiating this pattern one undeveloped goal will remain:</p> <ul style="list-style-type: none"> • G5 To support the decomposition of the top goal (G2) into further sub-goals, these goals must be developed to give a complete safety argument that the safety condition < safety -condition-x> holds for module <module-x> at location <loc-x> of code.
Implementation	<p>This pattern should be instantiated in a top-down approach. The number of goals and strategies in this argument are program-specific i.e., reflect the program structure and the information for the safety case construction is provided by the annotation inference. All texts highlighted in red are provided by the annotation inference and all texts in black are boiler-plate, i.e., can be reused in new contexts or applications without being changed much from the original except with the instantiation notation underneath them, e.g., J2 and J3.</p> <p>Possible Pitfalls</p> <ul style="list-style-type: none"> • The argument relies on the assumptions that the lists of variables and occurrences are complete. • The argument structure directly follows the course the annotation construction takes through the code and therefore the argument highly depends on the annotation inference system. Any error in annotation inference could cause doubts on the assurance provided because they only lead to unprovable VCs rather than directly observable evidence of unsafety.
Examples	As shown in Section 4.2, Figure 4.3
Known Uses	
Related Patterns	<p>Explaining the Safety Notion – This pattern can be used to derive G2</p> <p>Sufficiency of Safety Condition – This pattern can be used to decompose the undeveloped goals G5</p> <p>This pattern forms part of a program certification of auto-generated code pattern catalogue, which includes the following patterns:</p> <ul style="list-style-type: none"> ▪ Explaining the Safety Notion ▪ Sufficiency of Safety Condition

TABLE 4.9: Safety Case Pattern: Sufficiency of Safety Condition

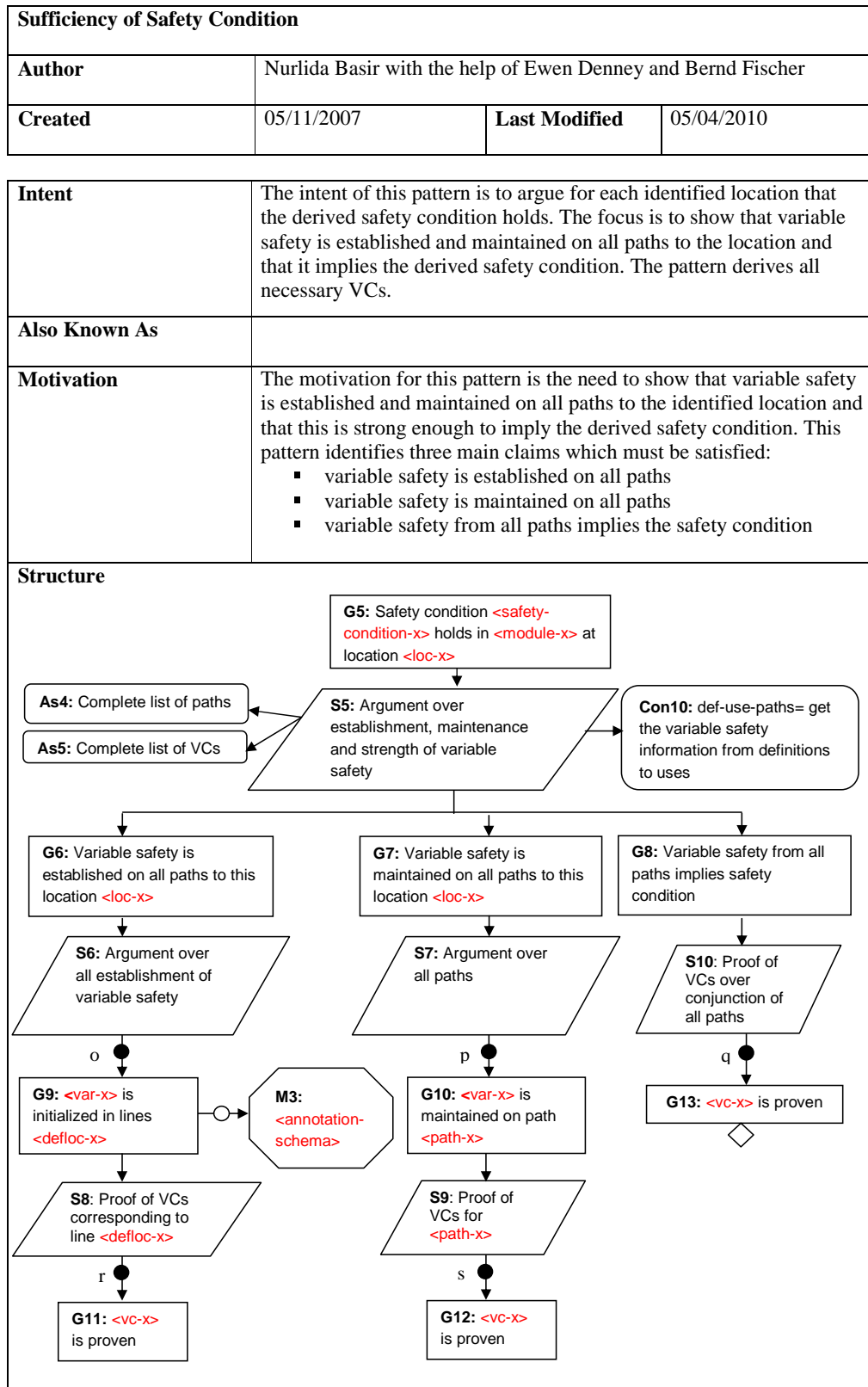


TABLE 4.10: Safety Case Pattern: Sufficiency of Safety Condition

Participants	G5	The overall objective of the argument – safety condition <safety-condition-x> holds in program <module-x> at location <loc-x> of code. Label < safety-condition-x > indicates the safety condition that should be shown to be valid and <loc-x> indicates the specific line of code in the program <module-x> .
	G6	This claim asserts that variable safety is established on all paths leading to this location <loc-x> .
	G7	This claim asserts that variable safety is maintained on all paths leading to this location <loc-x> .
	G8	This claim asserts that variable safety from all paths implies the safety condition i.e., <safety-condition-x> .
	G9 (o of)	This claim asserts that the variable safety <var-x> is defined in lines <defloc-x> of the code. The label <defloc-x> indicates the location in which the variable safety has been established. The number of goals is program-specific, i.e., depends on the number of definition locations identified by the annotation inference.
	G10 (p of)	This claim asserts that the variable <var-x> is maintained on path <path-x> . The number of goals is relies upon the number of paths (from definition location to use location or known as def-use chain).
	G11 (r of), G12 (s of), G13 (q of)	These claims assert that verification condition <vc> of the respective safety condition is proven sound. The number of goals depends on the number of constructed VCs.
	S5	The strategy presents the argument over the establishment, maintenance and strength of variable safety to support G5 .
	S6	Argues the establishment of variable safety on all paths.
	S7	Argues the maintenance of variable safety on all paths.
	S8	Argues proof of VCs corresponding to line <defloc-x> .
	S9	Argues proof of VCs for <path-x> .
	S10	Argues proof of VCs over conjunction of all paths.
	As4	Assumes that the argument iterates over the complete list of paths.
	As5	Assumes that the argument iterates over the complete list of VCs.

TABLE 4.11: Safety Case Pattern: Sufficiency of Safety Condition

	Con10	Describes the definition of def-use-paths i.e., get the safety information from the location where it is established to the location where it is used.
	M3 (optional)	Describes the annotation schema that is used during the annotation inference to identify the definition of the variable safety.
Collaborations	<ul style="list-style-type: none"> • As4 and As5 are statements that have to be relied upon in order to make the argument S5 that shows G5 valid. • Con10 describes the strategy on establishment and maintenance of variable safety. • G9 is necessary to support the implication of goal G10. 	
Applicability	This pattern is applicable for a program-specific argument, i.e., showing safety of all paths in the program to the specific location in the program.	
Consequences	<p>After instantiating this pattern a number of undeveloped goals will remain:</p> <ul style="list-style-type: none"> • G11, G12, G13 To support the decomposition of the top goal (G5) into sub-goals G6, G7, G8, these goals must be developed to give a complete safety argument that the safety condition <safety-condition-x> holds for <module-x> at location <loc-x> in the code. 	
Implementation	<p>This pattern should be instantiated in a top down approach. The safety case can be derived systematically and automatically as in “Safety of Program Variables” pattern. The inferred annotations are highly dependent on the actual program and the properties being proven. All texts highlighted in red are derived by the annotation inference and all texts in black are boiler-plate, i.e., can be reused in new contexts or applications without being changed from the original.</p> <p>Possible Pitfalls</p> <ul style="list-style-type: none"> • Attempting to decompose G7 without identifying the “defined” locations. • The argument is relies on the assumptions that the lists of paths and VCs are complete. • The argument is driven by the annotation inference algorithm. Any error in annotation inference could cause doubts on the assurance provided. 	
Examples	As shown in Section 4.2, Figure 4.4	
Known Uses		
Related Pattern	<p>Safety of Program Variables – This pattern can be used to derive G5</p> <p>This pattern forms part of a program certification of auto-generated code pattern catalogue, which includes the following patterns:</p> <ul style="list-style-type: none"> ▪ Explaining the Safety Notion ▪ Safety of Program Variable 	

Chapter 5

Requirement-Oriented Safety Cases

Chapter 5 describes an approach to systematically construct safety cases that show safety of a program wrt. a given safety requirement. This chapter also presents the argument patterns for the safety requirement. In contrast to Chapter 4, the structure of the safety argument here is driven by a user-specified set of formal safety assumptions and requirements rather than a fixed safety property, but the information for the safety case construction still comes from the same formal analysis of automatically generated code.

5.1 Introduction

In our work, we use Hoare-style program verification as the underlying verification approach. It can handle not only the safety properties considered in the previous chapter but other types of requirements as well. In this chapter, we exploit this and use the AutoCert system (i.e., the same system as in Chapter 4) to formally verify that the generated program is safe wrt. a given set of safety requirements and we construct safety cases that correspond to this more general verification style. We use two requirements on the components of a spacecraft guidance, navigation, and control (GN&C) subsystem as our running example here and in the next chapter. We thus next briefly describe the GN&C domain and the requirements before we discuss the difference between the property-oriented and the requirement-oriented verification approach.

Guidance, Navigation, and Control Systems. Spacecraft are typically decomposed into a number of different systems such as the power, thermal protection, or GN&C systems [114, 148]. Figure 5.1 illustrates the decomposition of the spacecraft into systems. The GN&C system is a necessary element of every spacecraft. Here, we

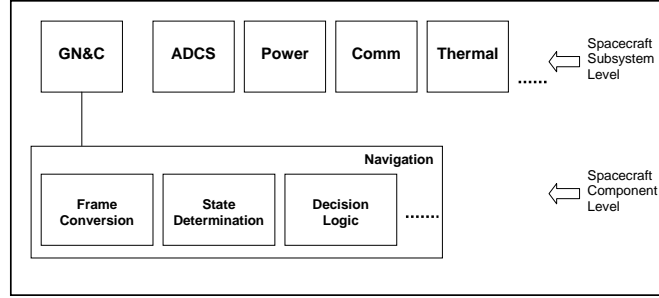


FIGURE 5.1: Spacecraft Decomposition (adapted from [114, 148])

focus on the navigation subsystem within the GN&C system. It is used to determine a spacecraft's orientation and position. The navigation subsystem contains, among of others, three sub-subsystems or components, a decision logic that computes a status value irrelevant to the requirements considered here, a frame conversion and a state determination component. The frame conversion first converts the frames of reference of the incoming signals from a vehicle-based coordinate system to an earth-based coordinate system. The transformations of the coordinate systems are usually done by converting quaternions to direction cosine matrices (DCMs), applying some matrix algebra, and then converting them back to quaternions [114, 148]. The state determination then performs the calculations to determine the vehicle state (position, attitude, attitude rate, etc.) from these signals. It is defined in terms of the relevant physical equations.

Code generators typically allow users to choose between “flat” and “structured” outputs, i.e., to choose whether the components' code is inlined into the main program, or wrapped to separate functions or modules. Here we ignore any internal structure that the navigation subsystem might have, and we assume that all components are “inlined”. We defer the handling of model structure to Chapter 6.

Formalization of the Requirements. In our example, we consider the navigation subsystem of the GN&C system as part of an overall spacecraft system. We assume that a hazard analysis of the overall system has identified the system hazards and that a fault tree analysis has attributed them to the corresponding subsystems, for example uncovering that the spacecraft can get into an unsafe state if the signal vel_2 is not representing a velocity measurement in the right frame (i.e., ECI). Based on the results of the fault tree analysis, the safety requirements on the subsystem can then be formulated; however, in this thesis we assume the safety requirements as given. Here and in the next chapter we concentrate on the following two safety requirements:

1. Signal $quat_3$ is a quaternion representing a transformation from the Earth-Centered Inertial (ECI) frame to the body fixed frame; and
2. Signal vel_2 is a velocity in the ECI frame.

Since we are working with a formal, logic-based analysis framework, we need to formalize these requirements using a domain theory, as follows:

- 1.' $quat_3 :: quat(ECI, Body)$
- 2.' $vel_2 :: vel(ECI)$

Here, ECI and $Body$ are constants denoting the respective frames, $quat$ and vel are functions denoting transformations of or quantities in those frames, and $::$ is a predicate that asserts that the signal represents a transformation between (resp. quantity in) the required frame(s).

Obviously, the formalization of the safety requirements is safety-relevant: a wrong formalization can invalidate the assurance provided by the proofs [26, 100]. It thus needs to be called out and justified in the safety case.

Here, the safety requirements result from the hazard analysis of the system. The safety requirements can also be considered as requirements on the system/software boundary. Therefore, the use of fault tree analysis is more suitable with respect to this situation than software fault tree analysis (SFTA). SFTA [94, 96, 97] is an approach to trace behavior in the code logic and identify whether a path exists that could cause hazardous output. SFTA is only concerned with the safety of the internal code logic which is not sufficient here as we also need to ensure that the safety requirements hold at a particular location in the program.

Safety Property vs Safety Requirement. A safety property is an invariant that needs to hold everywhere in the program. In contrast, a safety requirement is an assertion that needs to hold at particular location in the program (typically at the end), and thus has more of a liveness “flavor”. Safety properties typically also apply to all variables (or at least to all variables of a given type) in a program (cf. the definition of the initialization-before-use property), while safety requirements are usually formulated over a single variable, as in our running examples. In order to ensure that a program is safe, we usually need to show that it satisfies a set of both safety properties and safety requirements.

The validity of the safety requirement on each variable typically depends on the safety requirements of other variables (see example in Figure 5.2), in contrast to the situation for safety properties. Therefore, in the safety cases we need to construct an argument that reflects this dependency. Moreover, the requirements on some variables are not established by the program but are given as external assumptions which have to be relied on in showing program safety, and the safety case needs to call out this dependency as well.

In addition, the correct formalization of a safety property can be inspected and accepted off-line, i.e., independently of the programs, but the formalization of the safety requirement depends on the structure of the actual requirement itself. Therefore, we need to construct an argument that shows the transition from the informal representation to the formalized safety requirement. This argument helps in showing that the formal verification runs over the correct requirement, based on the right formula and variable, and thus provides a relevant proof of the program. However, the certification of safety requirements does not require a dedicated safety policy, because all safety-relevant information is reflected in the annotations, so the safety cases do not need to argue that its formalization of safety policy is adequate (cf. GP1 in the safety case pattern described in Section 4.3.1). Note that the annotations still remain untrusted, as any errors in them will make it impossible to prove the final VCs, and thus to show that the requirement holds.

5.2 Constructing Safety Cases for Demonstrating a Safety Requirement

We consider the safety requirements as given, and consider each violation of any given safety requirement as a hazard. The constructed safety case will argue that the safety requirement is in fact not violated at the given location. The core argument structure of these requirement-oriented safety cases is driven by the same formal program analysis as used for the property-oriented safety cases and thus similar. The main difference in the argument structure is in showing the validity of the safety requirement, i.e., in terms of formalization of the requirement, and in showing the dependency on the safety requirements of other variables. In particular, the validity of each safety requirement typically relies on the validity and consistency of a given set of external assumptions in the input variables, which is not the case in the property-oriented safety cases.

Similar to the overall approach in Chapter 4, we construct a generic, requirement-oriented safety case that makes explicit the formal and informal reasoning principles, and reveals the top-level assumptions and external dependencies that must be taken into account in demonstrating program safety wrt. a given safety requirement. We instantiate the generic safety case to code generated for NASA's Project Constellation, which uses Real-Time Workshop for its GN&C systems, and focus on the first requirement given on page 78 as an example, but our framework can handle other safety requirements as well. We again use the GSN [86] as technique to explicitly represent the linkage between evidence (i.e., formal proofs) and safety claims.

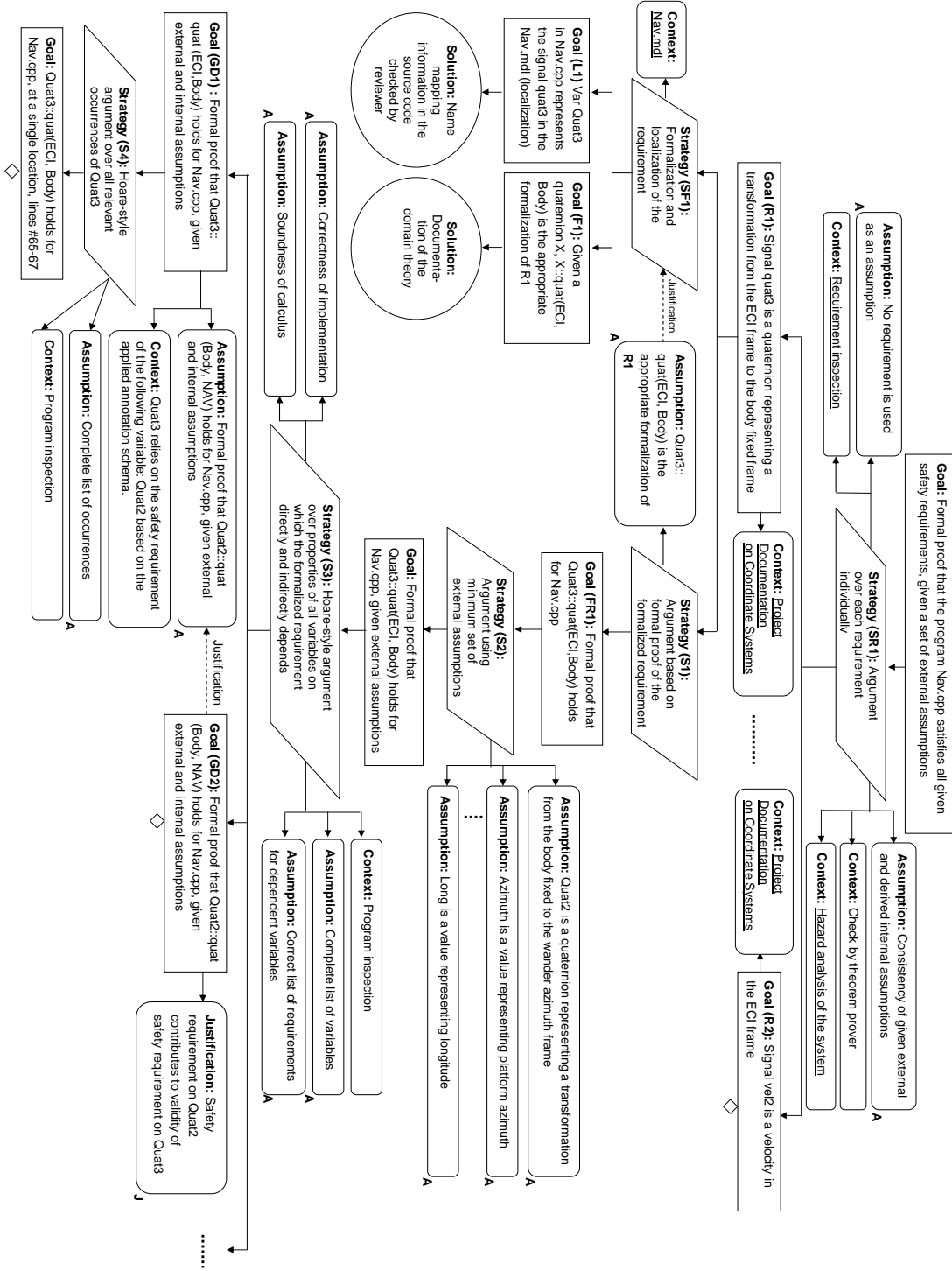


FIGURE 5.2: Arguing over Formalization of Safety Requirement

5.2.1 Arguing over Formalization of Safety Requirement

Figure 5.2 shows a safety case that argues over safety of the program wrt. all safety requirements as identified during the hazard analysis of the overall navigation system. The main difference in the argument of the safety requirements in comparison to the safety property is the need to argue over the formalization of the requirement and the properties of all variables on which the formalized requirement relies upon. However, the overall argument is still based on the Hoare-style argument, as in the property-oriented safety case.

The requirement-oriented safety case starts with the primary safety goal that the program (i.e., Nav.cpp) satisfies all safety requirements, given a set of external assumptions on the input variables, by providing formal proof as evidence. The key argument strategy here is to argue separately over each individual requirement that contributes to the program safety. The additional information that is required for the strategy to be understood and valid is identified and explained. This concerns the independent validity of the safety requirements and the logical consistency of the given external and any derived internal assumptions, which are both key to the strategy of arguing the requirements independently. We thus assume that no safety requirement is available for use as a (logical) assumption in the safety proofs, which prevents vacuous proofs based on mutually recursive dependencies between requirements and assumptions. We further assume that the given and derived assumptions together are consistent, again to prevent vacuous proofs. Each of these assumptions on the strategy is described by a context (e.g., the consistency of the assumptions can be checked by theorem prover and the use of the requirements by a requirement inspection). Context nodes with hyperlinks outline additional evidence in the form of documents, containing, for example, a detailed description of the system and requirement, and also the result of the hazard analysis. As a result of this strategy we now get as many subgoals as there are safety requirements that have been identified during the hazard analysis. Here we focus on the goal (R1) corresponding to the first requirement, i.e., that $quat_3$ is a quaternion representing a transformation from the Earth-Centered Inertial (ECI) frame to the body fixed frame.

The argument over each individual requirement is then split into two branches, first showing the correct formalization and localization of the requirement, and second, arguing over a formal proof of the requirement. The second branch is in some sense the main branch, since its argument structure is close to the program. Here, the first branch is a backing evidence to the second branch, which could be represented as an away goal. However, we again choose to represent the argument as shown in Figure 5.2 since the validity of the formal proofs depends crucially on the correct formalization and localization of the requirement. The first branch describes the transition from the informal level to a formalized safety requirement. This argument helps in showing that the formal verification runs over the correct requirement, based on the right formula and variable, and

thus provides a relevant proof of the program. It also justifies that the identified hazard E1 (see Figure 3.4, page 53) has been controlled. We use an explicit strategy to describe this transition, which spawns two subgoals. The first subgoal (F1) demonstrates that the formal proof is based on an appropriate formalization of the requirement, and the safety case points to the documentation of the logical domain theory as evidence of this. The second subgoal (L1) “glues together” the model level and the code level, which describes the localization of the requirement, i.e., the mapping process between the variable name in the program to the corresponding signal in the model. The correctness of the name mapping information can easily be checked in the source code by the reviewer.

Once the correct formalization and localization of the requirement has been shown, the further argument is to show that the formalized requirement (e.g., `Quat3::quat(ECI, Body)`) is the appropriate formalization of R1) holds for `Nav.cpp`. This can be achieved by an argument based on a formal proof of the safety requirement using the minimum set of external certification assumptions relevant to the requirement. Since not all given assumptions of the program are used in deriving the proofs of each requirement, a list of the used external assumptions is given with each requirement. The purpose of the argument is to justify that all assumptions that are relevant and sufficient in showing the program safety wrt. a given requirement have been considered (i.e., correct list of external assumptions). This also helps in identifying external assumptions that are entirely irrelevant, i.e., do not contribute to any of the given safety requirements.

Assuming that the argument has correctly identified all relevant external assumptions related to the formalized requirement, the further argument is to show that relevant properties of all variables on which the formalized requirement directly or indirectly depends, are valid. Here, the argument strategy relies on the assumptions that the calculus is sound, that its implementation is correct, that no variables are missing, and that the list of the dependent variables given is correct. The latter two assumptions (i.e., no missing variables and correct list of dependent variable) can be checked easily by the program inspection, which thus eliminate doubts about the incomplete program coverage (i.e., E2, E2.1, E2.2 and E2.3 in Figure 3.4, page 53) of the program verification. However, they are also discharged implicitly, along the way of the Hoare-style argument, since any omissions and errors in these lists will lead to unprovable VCs (assuming the soundness of the calculus and the correctness of its implementation). Note that the number of subgoals of the strategy S3 corresponds to the number of the variables on which the requirement directly and indirectly depends. In our example, *Quat3* is the main variable of the given safety requirement R1, while *Quat2* is a variable on which *Quat3* relies in showing its requirement holds in the program. Due to the dependency relation between *Quat3* and *Quat2*, any inconsistency in showing the safety requirement of *Quat2* can invalidate the argument showing the safety requirement on *Quat3*, which is described as a justification that is attached to the goal GD2. Note that in this case, the safety requirement on *Quat2* required in GD2 is actually given as external

assumption (i.e., to the system) to the previous strategy S2, so the argument below GD2 can appeal to this and only needs to show that the program maintains it through to the relevant occurrences. Subsequently, the safety requirement for each individual variable, for example *Quat3*, is then shown by arguing over its all relevance occurrences in the program, *Nav.cpp*. This reduces the original safety argument to a program verification problem, showing that a specific property holds of a specific locations in the program. Note that the number of subgoals of this strategy is corresponds to the number of variable occurrences as identified during the course the annotation construction takes through the program. The further argument is shown in the next section.

5.2.2 Sufficiency of Safety Conditions for Individual Variables

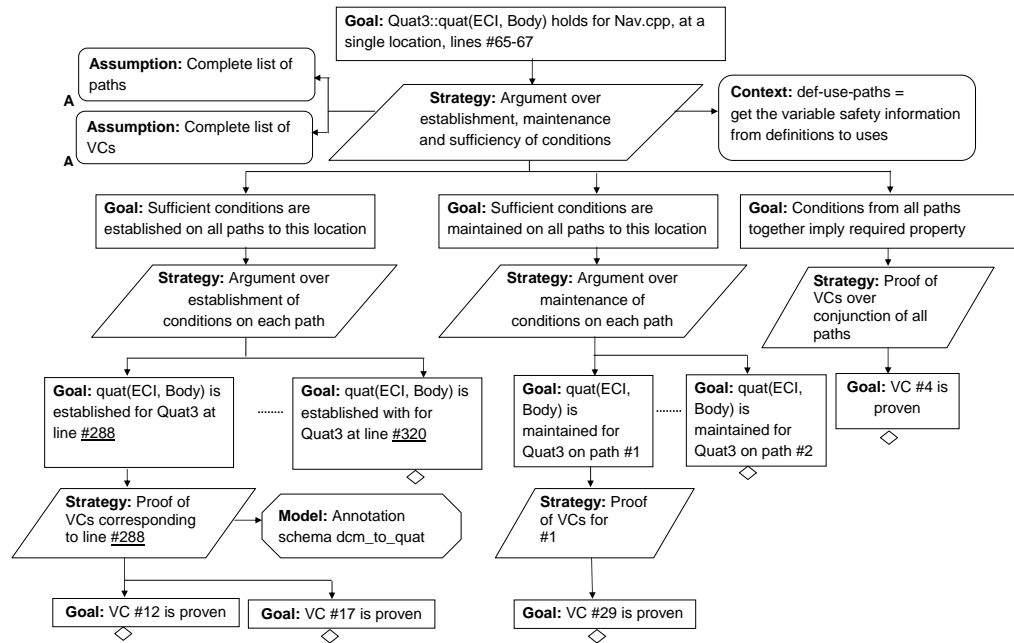


FIGURE 5.3: Arguing over Sufficiency of Safety Conditions for Individual Variables

The further argument for showing the safety requirement for each variable (i.e., *Quat3* and *Quat2*) is focused on showing that sufficient conditions on the variables are established and maintained on all paths to the location and that the conditions from all paths together establish the stated safety requirement. Typically, each path establishes the complete safety requirement, so that the last goal becomes trivial, but this is not necessarily the case. Figure 5.3 shows an example of a safety case that argues safety of variable *Quat3* in *Nav.cpp*. We concentrate on *Quat3* as an example as the same underlying argument structure can be applied for *Quat2* as well. The overall argument structure for this safety case follows the “Sufficiency of Safety Condition” pattern as defined in Section 4.3.3. The safety case can be derived systematically and automatically as the argument structure directly reflects the course the annotation inference has taken

through the code, as already described in Section 4.3.3.

The argument for the variable *Quat3* proceeds accordingly, as described in the pattern, which thus leads to a number of VCs. Here, the VCs are used to demonstrate that the conditions are established and maintained on all paths leading to the appropriate location and that the conditions from all paths imply the safety requirement. If (and only if) all VCs can be shown to hold, then the requirement holds for the given location. Each branch thus ends again with a goal to prove a set of VCs, which reduces the program verification argument to a purely logical proof argument.

5.3 Safety Case Patterns for Requirement-Oriented Safety Cases

Similar to the approach taken for safety properties, we can now define safety case patterns that generalize the structure of the requirement-oriented safety case. The patterns are generic and customizable for arguing safety of program wrt. all given safety requirements (i.e., for arguing that the software satisfies all given safety requirements). The information for the safety case construction in these patterns can be derived by a formal program analysis. Some of the texts in the patterns are fixed and some of them need to be further instantiated.

5.3.1 Safety Case Pattern: Formalization of Safety Requirement

The pattern generalizes the safety case for showing the safety of the program wrt. all given requirements as identified during the hazard analysis of the overall system. The key argument strategy is to argue program safety wrt. each individual safety requirement by showing that the formalization and localization of the requirement is correct and by providing formal proofs as evidence. The structure of the argument remains the same as shown in the example in Figure 5.2, but we generalize the contents of the safety case nodes. The argument pattern can be customized for other programs and safety requirements by changing certain information labelled in $\langle \rangle$, which can be derived from the formal program analysis such as the information about the safety requirement (i.e., $\langle \text{req-x} \rangle$) and its formal representation (i.e., $\langle \text{formalreq-x} \rangle$), the program (i.e., $\langle \text{module-x} \rangle$) and the safety requirement to be shown for the program (i.e., $\langle \text{formalproperty-x} \rangle$). The contents of some of the pattern nodes are fixed (i.e., can be reused in new contexts or applications) but some of them need to be further instantiated (labelled with a triangle \triangle) and further development (labelled with a diamond \diamond). The link model with \bullet describes a node that can be solved by zero or more instances. The subgoals and strategies here are requirement-specific (i.e., reflect the number of the safety requirements as identified during the hazard analysis of the system). Tables 5.1

to 5.5 describe in details the safety case pattern for arguing safety of program wrt. all given safety requirements.

5.3.2 Safety Case Pattern: Sufficiency of Safety Conditions for Individual Variables

The motivation for the argument on individual variables is the need to show that sufficient conditions on each variable are established and maintained on all paths to the identified location and that this is strong enough to imply the safety requirement. This pattern identifies three main claims which must be satisfied, i.e., the conditions are established on all paths, the conditions are maintained on all paths, and the conditions from all paths imply the safety requirement. The overall argument structure here follows the same lines of the safety case pattern as described in Section 4.3.3 with small modifications; in particular, the notion of safety condition is replaced by that of safety requirement.

5.4 Conclusions

In order to ensure that the program is safe, we usually need to show that the program satisfies both, general safety properties and specific safety requirements. In the previous chapter, we have developed safety cases that argue along safety of the generated code wrt. a given safety property. In addition, in this chapter, we have constructed safety cases for showing safety of the generated code wrt. a set of externally given safety requirements. We use the same underlying Hoare-style program verification as in Chapter 4 to verify the program safety. Here, the formal program verification analyzes the system structure on the code level to identify where the requirements are ultimately established, and so checks the code, providing independent assurance. We have also constructed a safety case pattern that is generic wrt. a given set of safety requirements.

The overall argument of the requirement-oriented safety cases is focused on showing the correct formalization of the safety requirement (i.e., the transition from the informal representation of the requirement to the formalized safety requirement) and the validity of the safety requirements for each dependent variable. We make explicit the formal and informal reasoning principles, and reveal the top-level assumptions and external dependencies (i.e., between the requirement and the given external assumptions) that must be taken into account in demonstrating the program safety, and use formal proofs as evidence. We illustrated our approach using the verification of a safety requirement for a spacecraft navigation system that was generated from a Simulink model by Real-Time Workshop [4]. However, since each system safety requirement induces a slice of the system architecture, it is also important to identify how the system safety requirements are

broken down into component requirements, and where they are ultimately established. Therefore, in the next chapter, we describe an approach to construct safety cases that argue along the hierarchical structure of systems in model-based development.

TABLE 5.1: Safety Case Pattern: Formalization of Safety Requirement

Formalization of Safety Requirement			
Author	Nurlida Basir with the help of Ewen Denney and Bernd Fischer		
Created	03/08/2009	Last Modified	05/04/2010
Intent	The intent of this pattern is to show the safety of program wrt. all given safety requirements by providing formal proof as evidence.		
Also Known As			
Motivation	<p>The motivation for this pattern is the need to show that the program satisfies all given safety requirements (as identified during the hazard analysis of the system). The argument considers each requirement individually. This pattern breaks the argument for each requirement into two reduction steps:</p> <ul style="list-style-type: none"> ▪ formalization and localization of the requirement ▪ formal proof that the requirement holds after each execution of the program 		

TABLE 5.2: Safety Case Pattern: Formalization of Safety Requirement

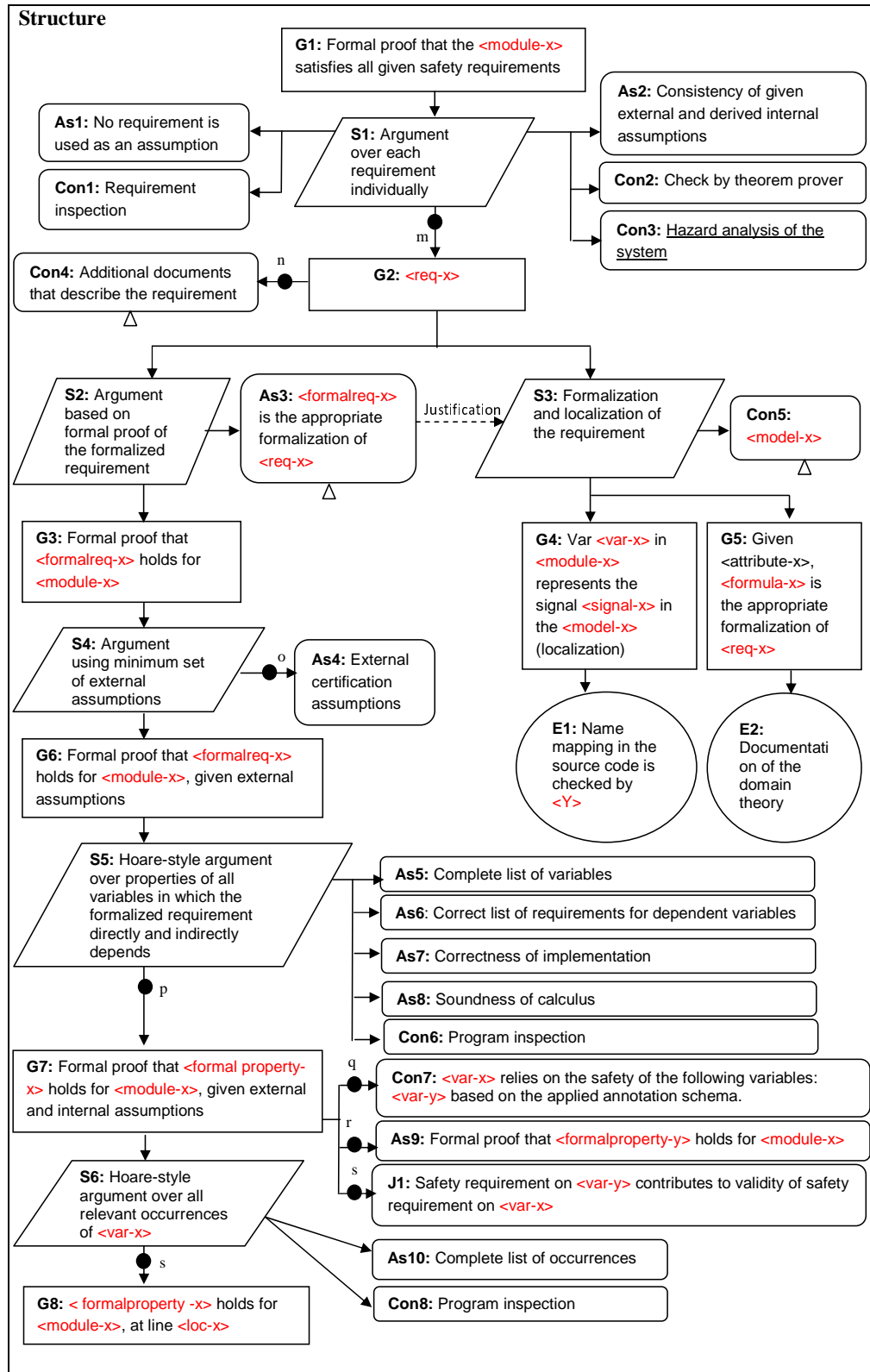


TABLE 5.3: Safety Case Pattern: Formalization of Safety Requirement

Participants	G1	The overall objective of the argument is to show that the program <module-x> satisfies all given safety requirements. Label <module-x> indicates the program that should be shown to be safe.
	G2 (m of)	The goal shows the safety requirement <req-x> of the program that should be satisfied. The number of goals corresponds to the number of safety requirements as identified during the hazard analysis.
	G3	This claim asserts that the formal requirement <formalreq-x> holds after each execution of module <module-x> .
	G4	This claim asserts that the variable <var-x> is the appropriate variable to represent the signal <signal-x> in the system model <model-x> .
	G5	This claim asserts that the given formalization of the requirement <formula-x> is correct.
	G6	This claim asserts that the formal requirement <formalreq-x> holds for module <module-x> , given the external certification assumptions.
	G7 (p of)	This claim asserts that the properties of the variable <var-x> , on which the formalized requirement <formalreq-x> depends hold in the program <module-x> . The number of G7 goals corresponds to the number of dependent variables.
	G8 (s of)	This claim asserts that the requirement property <formalproperty-x> holds for module <module-x> at line <loc-x> of the code. The number of goals is program-specific i.e., reflects the number of necessary occurrences of the variable in the program structure.
	S1	Decomposes the program safety to validity of each individual safety requirement relevant to the program.
	S2	The strategy presents the argument based on a formal proof of the formalized requirement.
	S3	The strategy presents the argument over the formalization and localization of the requirement.
	S4	Argues program safety using minimal set of given external certification assumptions.
	S5	The strategy presents the Hoare-style argument. The main focus of the argument is to argue over properties of all variables, on which the formalized requirement depends.

TABLE 5.4: Safety Case Pattern: Formalization of Safety Requirement

	S6	The strategy present the Hoare-style argument over all relevant occurrences of each variable <var-x>
	As1	Assumption that no requirement is used as a certification assumption.
	As2	Assumes consistency of given external and derived internal assumptions
	As3	Assumption that the <formalreq-x> is the appropriate formalization of the requirement <req-x> .
	As4 (o of)	Assumes complete list of external certification assumptions.
	As5	Assumes complete list of variables.
	As6	Assumes correct list of requirements for dependent variables.
	As7	Assumes that the calculus is sound.
	As8	Assumes that the implementation is correct.
	As9 (r of)	Assumes the requirement of the dependent variable holds for the program.
	As10	Assumes complete list of occurrences.
	J1 (s of)	Justifies that safety requirement on dependent variable <var-y> contributes to validity of the main safety requirement <var-x>
	Con1	Describe the correctness of assumption As1 can be shown by software requirement inspection and thus justified the strategy S1 .
	Con2	Describe the correctness of assumption As2 can be checked by a theorem prover and thus justified the strategy S1 .
	Con3	Provides link to the hazard analysis results of the system.
	Con4 (n of)	Any supporting documents that help in describing the requirement.
	Con5	Describes the model <model-x> of the program or module.
	Con6, Con8	Describes that the complete list of variables and occurrences can be checked by program inspection.
	Con7 (q of)	Describes the list of dependent variables <var-y> of the requirement.
	E1	Describes the evidence to prove the correctness of the requirement localization (e.g., name mapping in the source code is checked by <Y>). <Y> indicates the name of person or expert that checks the name mapping.
	E2	Describes the evidence to prove the correctness of the requirement formalization (e.g., documentation of the domain theory).

TABLE 5.5: Safety Case Pattern: Formalization of Safety Requirement

Collaborations	<ul style="list-style-type: none"> Assurance on As1 and As2 are necessary to support the implication of goal G2. Con1 is necessary in providing assurance on As1. Con2 is necessary in providing assurance on As2. As3 is statement that has to be relied upon in order to make the argument that G3 is valid. As4 is statement that has to be relied upon in order to make the argument that G6 is valid. As5, As6, As7 and As8 are statements that have to be relied upon in order to make the argument that G7 is valid. Con6 is necessary in providing assurance on As5 and As6 are valid, and Con8 is necessary in providing assurance As10 is valid. As9 is statement that has to be relied upon in order to make the argument that G7 is valid. The strategy S6 relies on the assumption As10 is valid.
Applicability	This pattern is applicable for arguing that the program satisfies each individual safety requirement of the program; it also shows the formalization of the requirement and the dependency of variables in the program.
Consequences	<p>After instantiating this pattern a number of undeveloped goals will remain:</p> <ul style="list-style-type: none"> G8 To support the claim that the module or program <module-x> satisfies all given safety requirements, further argument on showing that the requirement property <formalproperty-x> holds for module <module-x> at line <loc-x> of code should be further developed.
Implementation	<p>This pattern should be instantiated in top down approach. The number of subgoals (G2) is corresponds to the number of safety requirements as identified during the hazard analysis of the overall system. Texts highlighted in red are provided by a formal analysis of the program, model and requirement document, while all texts in black are boiler-plate i.e., can be reused in new contexts or applications without being changed much from the original.</p> <p>Possible Pitfalls</p> <ul style="list-style-type: none"> Correctness of the argument is relies upon the assumptions that no requirement is used as an assumption and assumptions are consistent. The further argument of G2 relies on correct formalization and localization of the requirement. If the requirement is incorrectly formalized, the overall proof argument is invalid. The further argument of G6 relies on the assumptions As5 and As6. G7 relies on the safety of its dependable variables.
Examples	As shown in Section 5.2 Figure 5.2
Known Uses	
Related Pattern	<p>Sufficiency of Safety Condition – This pattern can be used to decompose the undeveloped goals G8</p> <p>This pattern forms part of a program certification of auto-generated code pattern catalogue, which includes the following patterns:</p> <ul style="list-style-type: none"> Sufficiency of Safety Condition (Section 4.3.3 with some minor modifications) Explaining the Safety Notion (Chapter 4, Section 4.3.1 with some minor modifications)

Chapter 6

Architecture-Oriented Safety Cases

Chapter 6 presents an approach to systematically derive safety cases that argue along the hierarchical structure of systems in model-based development. The underlying analysis used in the previous chapter can also be used to recover the system structure and component hierarchy from the code, which is then turned into a safety case providing independent assurance of both code and model.

6.1 Introduction

Model-based development and automated code generation are increasingly used for actual production code, in particular in mathematical and engineering domains. However, since code generators are typically not qualified, there is no guarantee that their output is correct or even safe, and additional evidence of its safety is required. In the previous chapter, we have constructed requirement-oriented safety cases from information collected during the formal verification of the generated code. In this chapter, we extend that work to an approach to systematically derive safety cases that argue along the hierarchical structure of systems in model-based development. The safety cases are constructed mechanically using the same underlying formal analysis such as in the requirement-oriented safety cases. This is driven by a set of formal safety requirements on the model which express as logical formulas of the properties that the (software sub-) system's output signals must satisfy for the (overall) system to be safe. The original analysis is extended to recover and record the system structure and component hierarchy from the code. It identifies how the system safety requirements are broken down into component requirements, and where they are ultimately established, thus establishing a hierarchy of requirements that is aligned with the hierarchical model structure. The derived safety cases reflect the results of the analysis, and provide a high-level argument

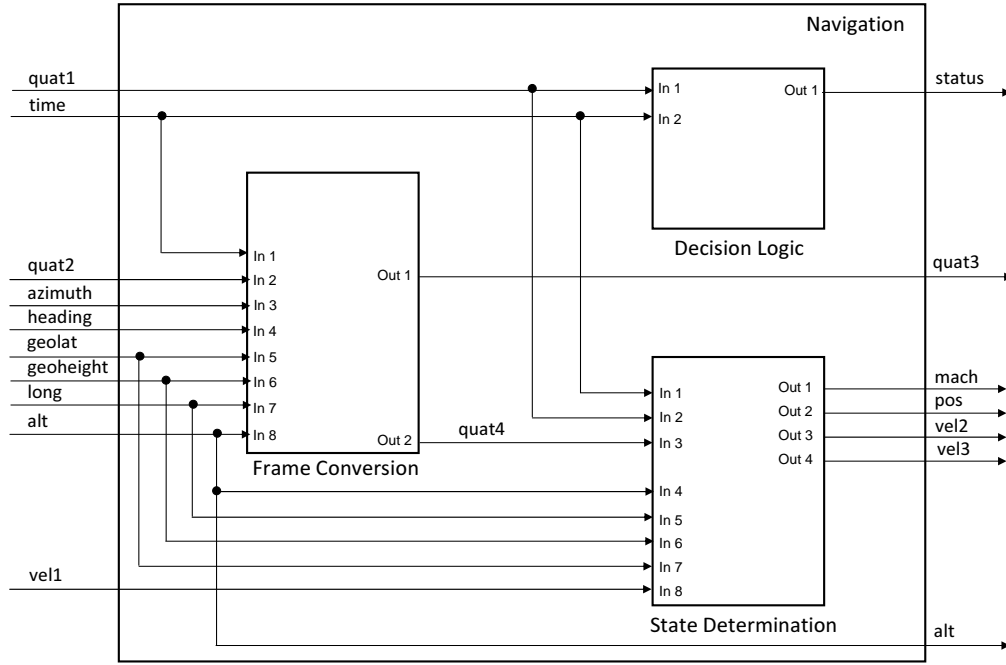


FIGURE 6.1: High-level Architecture of Navigation System

that traces the requirements on the model via the inferred system structure to the code, and hence, providing independent assurance of both code and model. We illustrate our work using the verification information of two safety requirements of a spacecraft navigation system that was generated from a Simulink model by Real-Time Workshop [4].

Navigation Subsystem. As already sketched in Chapter 5, the Navigation subsystem is comprised of three sub-subsystems or components, a decision logic, frame conversion and state determination. Here we consider the frame conversion and state determination components only, since the decision logic is irrelevant to our requirements. However, we left it in the model to show that the analysis identifies it as such. Note that there are no individual blocks within the Navigation subsystem, but only within the components and thus all computation happens there. Hence, we assume now (in contrast to Chapter 5) that the code generator keeps the hierarchical model structure in the code. We give a much simplified description of the subsystem; in particular, we have changed the names of components and signals. However, it should become clear that the navigation domain is challenging from a safety perspective, due to its complex and mathematical nature. In the following we simplify the description by denoting Navigation as a “system” rather than a “subsystem” and Frame Conversion and State Determination as “components” rather than “sub-subsystems”. However, both components are specified by independent models, so that we indeed work with a proper model hierarchy.

The Navigation system (cf. Figure 6.1 for the system architecture) takes several input signals, representing various physical quantities, and computes output signals represent-

ing quantities, such as Mach number, angular velocity, position in the Earth-Centered Inertial (ECI) frame, and so on. Signals are generally represented as floating point numbers or as quaternions and have an associated physical unit and/or frame of reference. However, the units and frames are usually not made explicit in the model, and instead are expressed informally in comments and identifier names.

Architecture Recovery. We concentrate on the same safety requirements (i.e., signal $quat_3$ is a quaternion representing a transformation from the the Earth-Centered Inertial (ECI) frame to the body fixed frame, and signal vel_2 is a velocity in the ECI frame) as described in Chapter 5, page 78. In order to certify these requirements on the Navigation system, and to build a comprehensible safety case, we need to know where in the system they are established, and which parts of the system contribute to them. Intuitively, we can see in the system architecture (cf. Figure 6.1) that the first requirement should be established by Frame Conversion, since the signal $quat_3$ comes straight out of that component (and similarly for vel_2 and State Determination in the case of the second requirement). However, this too simplistic view is not sufficient. First, without looking into the models corresponding to the components it is not clear whether the requirement is indeed established within a component, or simply passed through it (as it is for example the case with alt), and which of the component’s input signals (if any), or more precisely which assumptions on them, are used in establishing the requirement. However, simply expanding the component models destroys the hierarchical structure of the system. More importantly, the safety of the system ultimately depends on the safety of the code rather than the model, but because we cannot trust the code generator to translate the model correctly we cannot derive any trust from the model.

Instead, AutoCert analyzes the code and recovers the slice of the system architecture that is relevant to a given safety requirement. AutoCert records when the control flow based analysis enters respectively leaves a component (implemented by RTW as a parameter-free procedure), and then remove the part of the requirements-definition chain that is contained within the component. The key to obtaining precise architecture slices is to identify situations in which the control flow path just passes through a component, without encountering a definition. In these cases, we can ignore the component altogether. AutoCert then assembles the slices from the signals involved in the recorded requirements-definitions chains and from the retained component. Note that the recovered architecture slices are not based on the call graph (even though the analysis follows the control flow graph) but on the implicit dataflow. They are thus very similar to the also dataflow-based Simulink models from which the code was originally generated. The main difference is that the slices use the generated variable names (here simplified to capitalized versions of the corresponding signal names) while the model of course uses the signal names. This analysis is integrated into AutoCert’s annotation inference algorithm, and the safety cases we construct require this information as input.

Figure 6.2 shows the architecture slices recovered for both requirements. In both cases,

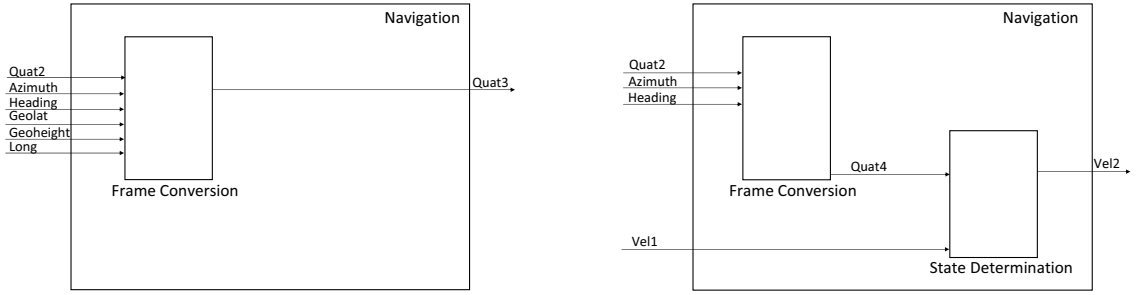


FIGURE 6.2: Architecture Slices Recovered for Example Requirements

the irrelevant Decision Logic component has been removed by the analysis. For the first requirement, it has further identified that $Quat_3$ is unaffected by the call to the State Determination procedure, and consequently removed that component as well. For the second requirement, the analysis has identified $Quat_4$ as the (global) variable through which the two components communicate. In addition, (although not shown in Figure 6.2), it has derived the property placed as an assumption on this variable by State Determination, i.e., $Quat_4 :: \text{quat}(\text{NED}, \text{Body})$. This becomes a subordinate requirement to the original safety requirement, reflecting the hierarchical model structure. The requirements hierarchy is completed by the assumptions placed on the variables i.e., Vel_1 and i.e., $Quat_2$ corresponding to the components' input signals.

The property derived for $Quat_4$ also becomes part of the interfaces of both components that are connected through this link, as assumption on the target end (i.e., State Determination) and as safety requirement on the source end (i.e., Frame Conversion). By regrouping the analysis results by component rather than by original safety requirement, we thus obtain full component interfaces. They give a complete functional specification of the component, including all assumptions, as far as it is required to satisfy the given system-level safety requirements. The interfaces also serve as starting point for verifying the components independently, hence allowing a compositional verification of the system which makes it easier to scale the approach to larger systems. Currently, we only consider nominal component behavior, but our approach could also be applied to the off-nominal case, provided that appropriate safety requirements for the off-nominal modes can be identified.

The recovered system architecture and requirements hierarchy already constitute a core safety argument: Navigation satisfies the safety requirement (2') if the components Frame Conversion and State Determination satisfy their respective interfaces, and the requirements for Vel_1 , $Quat_2$, and $Quat_4$ hold. This argument can serve as blueprint for a full-fledged safety case. In addition, the derived component interfaces serve as starting points for the construction of independent safety cases for the components, yielding a hierarchy of safety cases that is aligned with the system's hierarchy of models.

6.2 Constructing Safety Cases from the Formal Analysis of Hierarchical Systems

We now describe (in simplified form) the safety cases derived from the information provided by the formal analysis of hierarchical systems. Their root goal is to show that the system satisfies the given safety requirements. The safety cases represent knowledge about the system's architecture and tracing information between code and model. This structural information needs to be passed down to and processed by the formal verification phase, because it can directly influence the construction of the core argument of the safety case. We then describe the hierarchical structure of the safety cases and show how they are derived systematically from information uncovered by the analysis phase. The resulting safety cases provide a traceable safety argument that shows in particular where the code, subsystem, and system depend on any internal and external assumptions. As before, we use the GSN [86] as technique to explicitly represent the logical flow of the safety argument.

6.2.1 Arguing from System-Level Safety Requirements to Component-Level Safety Requirements

The key argument strategy remains the same as in the non-hierarchical case, i.e., we still argue over each individual requirement that contributes to the program safety. The top level argument of the safety case thus follows the same lines as the top level argument strategy (SR1) in Figure 5.2, page 81. However, here we focus on the goal (R2) corresponding to the second requirement, i.e., showing that vel_2 is a velocity in the ECI frame, because this induces a more interesting slice of the system architecture (cf. Figure 6.3).

As described in Section 5.2.1, the next step of the argument uses two strategies, first, arguing the transitions from the informal level to a formalized safety requirement (SF2) which spawns two subgoals and second, arguing over a formal proof of the requirement (S2) which leads to one subgoal. The first leg of the argument (i.e., SF2) is the backing evidence for the second leg of the argument (i.e., S2) which in turn could be represented as away goal. The first subgoal (F2) of strategy (SF2) demonstrates that the formal proof is based on an appropriate formalization of the requirement, and the second subgoal (L2) of strategy (SF2) combines the model level and the code level, which allows us to build a safety case for the model based on the analysis of the code. In particular, as discussed in the paragraph describing the architecture recovery (page 95), we need to show the mapping between the signal names used in the model and the corresponding variable names used in the source code, which cannot be recovered by our analysis but must be given externally. Here, the safety case points to the mapping information given in the source code, and that it has been checked by a reviewer, as evidence. In addition,

at this goal we also have to show the mapping between the model and code files, and in particular, in which code file the property formalized in (F2) has to be shown. In our example, the localization mapping is straightforward, but for larger systems this localization needs more evidence.

With the assumption that the results of (F2) and (L2) are the appropriate formalization of the requirement, we can now construct the subgoal (FR2) of the strategy (S2), which states that the fully formalized safety requirement $Vel_2:: vel$ (EC1) holds after execution of the code in `Nav.cpp`. This requirement eventually needs to be proven formally, which can be achieved by compositional verification based on the Navigation system architecture. However, at this level of abstraction, the safety case does not contain an argument based on the full formal proofs. Instead, and in contrast to Chapter 5, we use an argument based on the system architecture, or more precisely, on the recovered system architecture slices. It shows how the system-level requirements are broken down into the component-level requirements i.e., into properties of the part of the system that is relevant to satisfy the requirement (FR2). This is based on an assumption to the strategy that the formal analysis has identified all relevant components and signals.

We thus reduce (FR2) to a number of (delayed) subgoals for the components and signals in the architecture slice. For each component, we need to show that it satisfies the safety requirements specified in its inferred interface (cf. subgoals (C1) and (C2)). This induces a further assumption on the strategy, namely that the interface is strong enough to show the requirement (FR2). Delaying the subgoals allows us to reuse the component-level safety cases. This way, we achieve a hierarchical structure for the system safety case that mirrors the hierarchy embedded in the system architecture. If the system contains top-level blocks in addition to the components (which is not the case in our example), we need to reason about their properties as well. This is indicated by the dashed subgoal (TLB). For each variable representing a signal, we need to show that it satisfies the safety requirements derived by the analysis (cf. subgoals (S1) to (S5)). This guarantees that the components' assumptions are met.

These subgoals are delayed here as well, to keep the safety case compact. Their expanded structure again follows the lines of Chapter 4, and in particular uses the argumentation shown in Figure 4.4 (Tier III) of the safety case there with small modifications; in particular, the notion of safety condition needs to be replaced by that of safety requirement. Note that we make no distinction at this level between subgoals that are established by the components, such as (S2), and those that are reduced to assumptions about the system's input signals and thus have trivial formal proofs, e.g., (S4).



6.2.2 Arguing from Component-Level Safety Requirements to Source Code

In the next step of our hierarchical development, we argue about the safety of the components wrt. their identified interfaces. The component-level safety cases also argue about a set of requirements, but there are two significant differences to the system-level safety cases. First, the component-level requirements are already formalized, due to the use of the formal analysis, so that we do not need to argue about the safety of the formalization and localization any more. Second, the argument will generally go down to the level of the generated code, with the proofs of the VCs as evidence; obviously, however, another layer of hierarchy is introduced if a component contains further (sub-system) components.

Figure 6.4 shows the safety case for the Frame Conversion component. For each component, the strategy is to argue over each individual safety requirement stated in its interface. Here, we have two requirements, (FC1) and (FC2). They serve different purposes in the system-level safety case—(FC1) is used to discharge the (essentially identical) system-level goal (FR1) via (C1), while (FC2) is used to discharge the signal subgoal (S2)—but at the component level we treat them the same. We focus on (FC2) here.

The component interfaces also list the assumptions that the component itself makes about the environment. However, not all assumptions are used for all requirements, so we use an explicit strategy to argue only using the minimal set of external assumptions, i.e., those on the system's input signals. Note that the use of internal assumptions (e.g., on $Quat_4$), which have been identified as subgoals in the system-level safety case (cf. subgoal (S2) in Figure 6.3) will be made explicit further down in the component-level safety case.

The next strategy finally transitions from the safety argument to a program correctness proof, using a Hoare-style argument over all relevant occurrences of the variable. In this case, it leads to a single subgoal, proving that the safety requirement holds at the given source location. This is predicated on the assumption that the applied Hoare-calculus is sound, and that the VCG is implemented correctly, which need to be justified elsewhere (cf. Figure 4.2 Tier I of the safety case presented in Chapter 4). The structure of the Hoare-style argument is determined by the structure of the program. Since the rest of the safety case is constructed as described in Chapter 4 Section 4.2.4, we do not expand here the final goal any further. Showing the safety of the component is thus reduced to formally showing the validity of the VCs associated with each requirement in the interface: a program is considered safe wrt. a given safety requirement if proofs for the corresponding VCs can be found. If (and only if) all VCs can be shown to hold, then the property holds for the entire program. The argument for proving the VCs is described in Chapter 7.

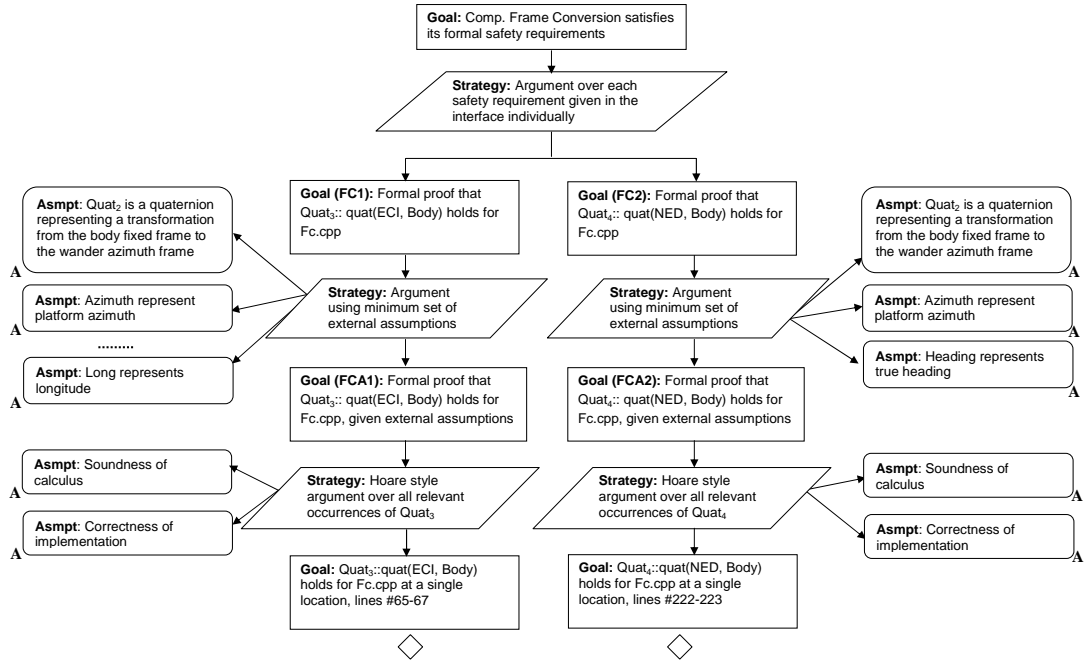


FIGURE 6.4: Component-level safety case for Frame Conversion

6.2.3 Combining System-Level and Component-Level Safety Cases

Splitting the argument into system-level and component-level makes it easier to follow the argumentation and allows us to factor out common sub-arguments, but in order to obtain a complete argument we need to combine the system-level safety case and the component-level safety cases. However, simply attaching the entire component-level safety cases to the corresponding component goals would introduce redundancies. Clearly, not every safety requirement on the system level relies on the full set of requirements established by the components, for example, (FR2) only uses the requirement derived for $Quat_4$ (i.e., goal (FC2) in Figure 6.4).

We thus replace each component goal only by the “branches” of the component-level safety case that are required; this information is available from the program analysis. For component goals that are shared between different requirements this will lead to an “unsharing”. For example, (C1) will be replaced by the branch rooted in (FC1) below (FR1) and by the one rooted in (FC2) below (FR2). However, common subgoals at the level of the Hoare-style argument (based on computations contributing to different requirements) can remain shared.

In addition to that, changes occur elsewhere in the system-level safety case. The assumptions to the architecture-based strategy solving (FR1) and (FR2) can be removed because the detailed argumentation in the component-level safety case provides the necessary evidence. Further, the subgoals associated with the system’s input signals (i.e., (S1) and (S3)–(S6)) can be removed because corresponding subgoals still appear as leaves

in the component-level safety case, where they are discharged by the assumptions. The subgoals on the connecting signals (here only (S2)) will be replaced by the root goals of the corresponding branches in the component-level safety case (i.e., (FC2)) at the appropriate position in the Hoare-style argument for the client component (i.e., State Determination).

6.3 Safety Case Patterns: Architecture-Oriented

We again define safety case patterns to generalize the structure of the safety cases, now for arguing along the hierarchical structure of systems in model-based development. The patterns are used as a way of abstracting the fundamental strategies from the architecture-oriented safety case. The hierarchical system structure argument represents the system structure and component hierarchy from the code, providing independent assurance of both code and model. In principle, the patterns described in the following subsections are generic as the general safety considerations at each tier are unchanged. Some of the texts are fixed and some of them need to be further instantiated.

6.3.1 Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements

The pattern generalizes the safety case for arguing how the system safety requirements are broken down into component requirements. The structure of the argument remains the same as shown in the example in Figure 6.3, but we generalize the contents of the safety case nodes. The argument pattern can be customized for other systems by changing certain information labelled in $\langle \rangle$. This information can be derived from a formal program analysis of the hierarchical system and also from the system model such as the information about the safety requirement (i.e., $\langle \text{req-x} \rangle$) and its formal representation (i.e., $\langle \text{formalreq-x} \rangle$) and information about the architecture slices recovered for the requirement. Tables 6.1 to 6.4 describe in details the safety case pattern for arguing the hierarchical decomposition of the requirement in the system.

6.3.2 Safety Case Pattern: Arguing from Component-Level Safety Requirements to Source Code

The pattern generalizes the safety case for arguing over each individual safety requirement of the component and demonstrates how the component safety requirements are moved down to the level of the code. The structure of the argument remains the same as shown in the example in Figure 6.4, but we generalize the contents of the safety case nodes. The argument pattern can be customized for other components by changing

certain information such as the information about the component (i.e., $\langle \text{comp-x} \rangle$), the program (i.e., $\langle \text{module-x} \rangle$) and information about the architecture slices recovered for the component requirement (i.e., $\langle \text{formalsignal-x} \rangle$). Tables 6.5 to 6.7 describe in details the safety case pattern for arguing the hierarchical decomposition of the requirement in the component.

6.4 Conclusions

With the increased use of model-based development in safety-critical applications, the integration of safety cases into such approaches has become important, especially in providing safety assurance in the system design. Hence, in this chapter, we have described an approach whereby the hierarchical structure of systems in model-based development drives the construction of architecture-oriented safety cases for the generated code. Here, the safety cases are constructed mechanically using information about the structure and component hierarchies recovered from the code by a formal program analysis. The safety cases thus provide independent assurance of both code and model. We show how the system safety requirements are broken down into component requirements and where they are ultimately established. We believe greater confidence in the assurance claim can be placed if the rationale behind the the transition from the system architecture and model to the program can be shown. We thus make an explicit argument over the correct transition from the model level representation to the source-level representation, including an argument over correctness of the formalization of the requirement. We show how the external assumptions on the system's input signals are used in establishing the safety of the program wrt. the given safety requirement. Like Rushby [126], we believe that a safe design will have ensured that the assumptions are valid. Currently, we only consider nominal component behavior, but our approach could also be applied to the off-nominal case, provided that appropriate safety requirements for the off-nominal modes can be identified. So far, we have illustrated our approach to flight code generated by Real-Time Workshop from hierarchical Simulink models. We are confident that the same approach can be applied to other modelling systems and generators as well.

TABLE 6.1: Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements

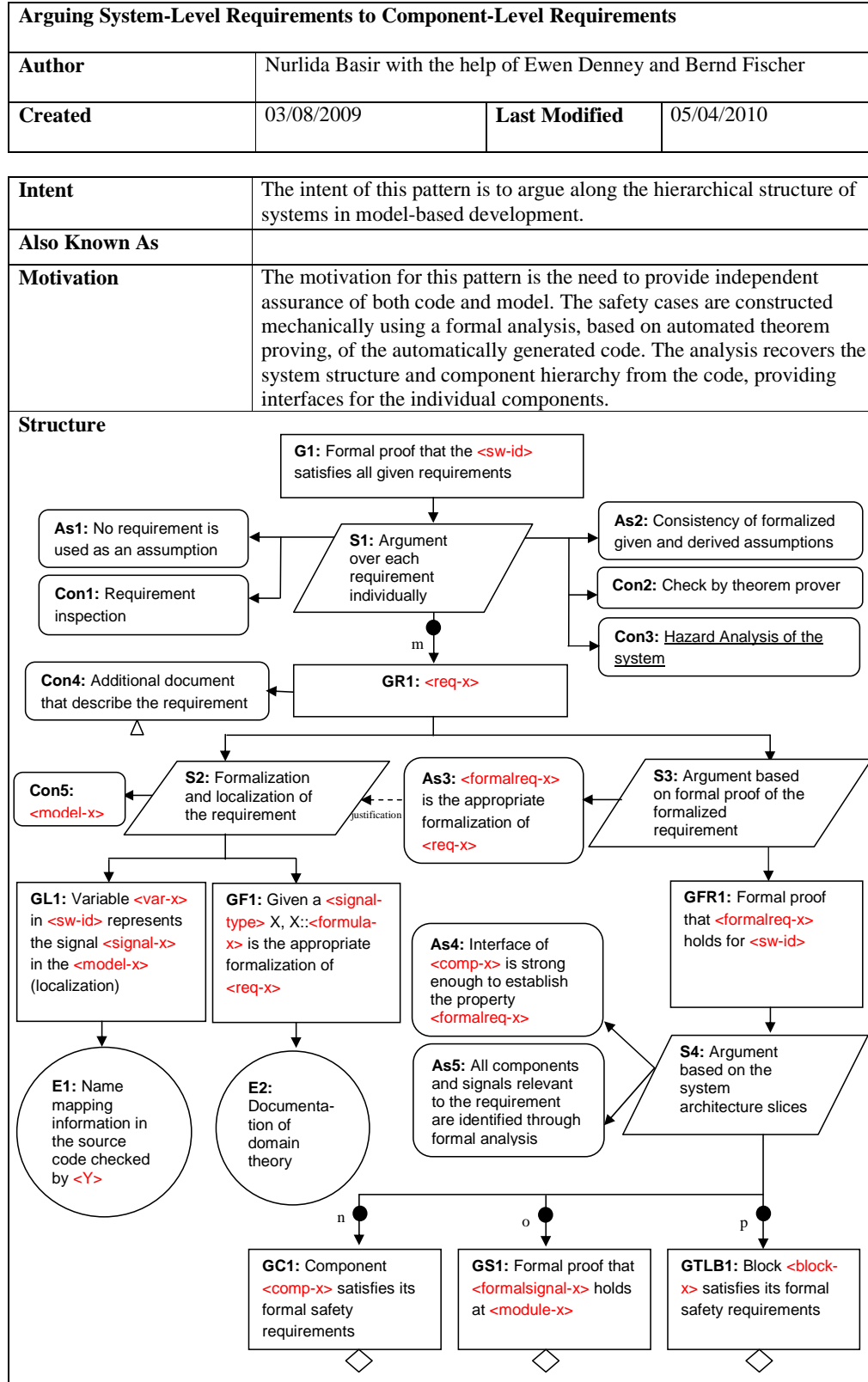


TABLE 6.2: Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements

Participants	G1	The overall objective of the argument is to show that software <sw-id> satisfies all given safety requirements. Label <sw-id> indicates the software that should be shown to be safe.
	GR1 (m of)	This claim asserts that the program satisfies the safety requirement <req-x> . The number of goals is relies upon the number of safety requirements as identified during the hazard analysis.
	GL1	This claim asserts that the given localization of the requirement is correct i.e., variable <var-x> in software <sw-id> represents the signal in the model <model-x> .
	GF1	This claim asserts that the given formalization of the requirement <req-x> is correct.
	GFR1	Formal proof that the formal requirement <formalreq-x> holds for software <sw-id> .
	GC1 (n of)	This claim asserts that the component <comp-x> satisfies its formal safety requirements. The number of goals corresponds to the number of components as recovered for the requirement.
	GS1 (o of)	This claim asserts that the formal proof for the formal safety requirement <formalsignal-x> holds at respective module <module-x> . Label <module-x> indicates the module that exists in the software. The number of goals is relies upon the number of signals as described in the system model.
	GTLB1 (p of)	This claim asserts that the top-level block <block-x> satisfies its formal safety requirements. The number of goals is relies upon the number of blocks as recovered for the requirement.
	S1	Decomposes the program safety to validity of each individual safety requirement relevant to the program.
	S2	The strategy presents the argument over the formalization and localization of the requirement.
	S3	The strategy presents the argument based on a formal proof of the formalized requirement.

TABLE 6.3: Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements

	S4	Argues the software safety by breaking down the argument into safety on each individual component, block and signal relevant to the software. In order to show that formal proof <formalreq-x> holds, an argument based on the system architecture slices is made.
	As1	Assumes there is no requirement is used as an assumption.
	As2	Assumes consistency of formalized given and derived assumptions.
	As3	Assumption that the <formalreq-x> is the appropriate formalization of the requirement <req-x> .
	As4	An assumption that interfaces of component <comp-x> is strong enough to establish the safety of the requirement <formalreq-x> .
	As5	An assumption that all components and signals relevant to the requirement have been identified through formal analysis.
	Con1	Describe the correctness of assumption As1 can be shown by software requirement inspection and thus justified the strategy S1 .
	Con2	Describe the correctness of assumption As2 can be checked by a theorem prover and thus justified the strategy S1 .
	Con3	Link to the hazard analysis results (used to define the safety requirements) of the overall system.
	Con4	Describes supporting document that can be used to explicitly describe the system and safety requirement.
	Con5	Describes the name of the model <model-x> that is used to construct the software <sw-id> .
	E1	The correctness and consistency of name mapping from model to source code can be checked by reviewer <Y> and provided as evidence.
	E2	Documentation of domain theory consists of formulae that can be used to formalize the requirement.
Collaborations	<ul style="list-style-type: none"> Assurance on As1 and As2 are necessary to support the implication of goal GR1. Con1 is necessary in providing assurance on As1. Con2 is necessary in providing assurance on As2. As3 is statement that has to be relied upon in order to make the argument that GFR1 is valid. Correctness of GL1 and GF1 are essential in providing assurance 	

TABLE 6.4: Safety Case Pattern: Arguing System-Level Safety Requirements to Component-Level Safety Requirements

	<p>on correct argument of GFR1.</p> <ul style="list-style-type: none"> • As4 and As5 are statements that have to be relied upon in order to support the strategy S4 in showing GFR1 is valid.
Applicability	<p>This pattern is applicable for arguing how the system safety requirements are broken down into component requirements, and where they are ultimately established, thus establishing a hierarchy of requirements that is aligned with the hierarchical model structure.</p>
Consequences	<p>After instantiating this pattern a number of undeveloped goals will remain:</p> <ul style="list-style-type: none"> • GC1, GS1 and GTLB1 To show that the software satisfies the given requirement, these sub-goals must be developed to give a complete argument.
Implementation	<p>This pattern should be instantiated in top down approach. The number of sub-goals (GR1) corresponds to the number of safety requirements identified during the system hazard analysis. The information for the argument is driven by the formal program analysis and by tracing the system architecture from the model to the code. All texts highlighted in red are provided by the formal analysis and provided by the specified information about the system and model, while all texts in black are boiler-plate i.e., can be reused in new contexts or applications without being changed much from the original.</p> <p>Possible Pitfalls</p> <ul style="list-style-type: none"> • The argument relies upon the assumptions that no requirement is used as an assumption and the consistency of formalized given and derived assumptions. • The further argument on GFR1 relies on correct formalization and localization of the requirement. If the requirement is incorrectly formalized, the overall proof argument might be invalid. • The further argument of GFR1 relies on the assumptions As3, As4 and As5.
Examples	As shown in Section 6.3 Figure 6.3
Known Uses	
Related Pattern	<p>Component-Level Requirements to Source Code</p> <p>– This pattern can be used to break down the undeveloped goals GC1 and also GS1 and GTLB1 (sub-arguments)</p> <p>This pattern forms part of a hierarchical safety cases, which includes the following patterns:</p> <ul style="list-style-type: none"> ▪ Component-Level Requirements to Source Code ▪ Sufficiency of Safety Condition (Chapter 4, Section 4.3.3 with some minor modifications) ▪ Explaining the Safety Notion (Chapter 4, Section 4.3.1 with some minor modifications)

TABLE 6.5: Safety Case Pattern: Arguing Component-Level Safety Requirements to Source Code

Arguing Component-Level Requirements to Source Code			
Author	Nurlida Basir with the help of Ewen Denney and Bernd Fischer		
Created	03/08/2009	Last Modified	05/04/2010
Intent	The intent of this pattern is to argue the safety of a component wrt. the safety requirements given in the interface.		
Also Known As			
Motivation	The motivation for this pattern is the need to provide independent assurance on the transition from the component-level of the system to the source-level of the generated code, with the proofs of the VCs as evidence.		
Structure	<pre> graph TD GC1[GC1: Component <comp-x> satisfies its formal safety requirements] --> S5[/S5: Argument over each safety requirement given in the interface individually/] S5 -- q --> GFC1[GFC1: Formal proof that <formal signal-x> holds for <module-x>] GFC1 --> S6[/S6: Argument using minimum set of external assumptions/] S6 -- r --> As6[As6: External certification assumptions] S6 --> GFCA1[GFCA1: Formal proof that <formal signal-x> holds for <module-x>, given external assumptions] GFCA1 --> S7[/S7: Hoare-style argument over all relevant occurrences of <var-x>/] S7 --> As7[As7: Soundness of calculus] S7 --> As8[As8: Correctness of implementation] S7 -- s --> GFCO1[GFCO1: <formalsignal-x> holds for <module-x> at location, lines <loc-x>] GFCO1 --> End{ } </pre> <p>The flowchart illustrates the structure of the safety case pattern. It begins with a goal node GC1: Component <i><comp-x></i> satisfies its formal safety requirements. This leads to a strategy node S5: Argument over each safety requirement given in the interface individually. A goal node GFC1: Formal proof that <i><formal signal-x></i> holds for <i><module-x></i> follows. Then, a strategy node S6: Argument using minimum set of external assumptions is shown. This node has a relationship <i>r</i> to an assumption node As6: External certification assumptions (marked with a triangle) and leads to a goal node GFCA1: Formal proof that <i><formal signal-x></i> holds for <i><module-x></i>, given external assumptions. This leads to a strategy node S7: Hoare-style argument over all relevant occurrences of <i><var-x></i>. This node has relationships to assumption nodes As7: Soundness of calculus and As8: Correctness of implementation. Finally, a goal node GFCO1: <i><formalsignal-x></i> holds for <i><module-x></i> at location, lines <i><loc-x></i> is reached via relationship <i>s</i>, leading to a diamond-shaped end node.</p>		

TABLE 6.6: Safety Case Pattern: Arguing Component-Level Safety Requirements to Source Code

Participants	GC1	The overall objective of the argument is to show that component <comp-x> satisfies its formal safety requirements. Label <comp-x> indicates the component of the system.
	GFC1 (q of)	This claim asserts that the formal safety requirement given in the interface <formalsignal-x> holds for module <module-x> by providing formal proof as evidence. Here, a component is equivalence to a module in the program. The number of goals is relies upon the number of component's safety requirements that should be shown hold.
	GFCA1	This claim asserts that the formal safety requirement given in the interface <formalsignal-x> holds for module <module-x> , using the given external assumptions.
	GFCO1 (s of)	This claim asserts that the formal safety requirement given in the interface <formalsignal-x> holds for module <module-x> at location, lines <loc-x> , by providing formal proof as evidence. The number of goals is program-specific i.e., reflect to the number of necessary occurrences of the variable in the program structure.
	S5	Argues over each individual safety requirement as stated in component interface.
	S6	Argues using the minimal set of external (i.e., on the system's input signals) assumptions.
	S7	Argues safety of program to safety on all relevant occurrences of variable.
	As6 (r of)	List of external certification assumptions that are relevant to the requirement.
	As7	Assumes soundness of the Hoare-calculus.
	As8	Assumes the VCG is implemented correctly.
Collaborations	<ul style="list-style-type: none"> Assurance on As6 is necessary to support the implication of goal GFCA1. As7 and As8 are statements that have to be relied upon in order to show that GFCO1 is valid. 	

TABLE 6.7: Safety Case Pattern: Arguing Component-Level Safety Requirements to Source Code

Applicability	This pattern is applicable for arguing over each individual safety requirement stated in component interface.
Consequences	After instantiating this pattern, the undeveloped goals (GFCO1) will remain. Showing the safety of the component is thus reduced to formally showing the validity of the VCs associated with each requirement in the component interface.
Implementation	<p>This pattern should be instantiated in top down approach. The number of sub-goals (GFC1) here corresponds to the number of safety requirements given in the interface of the component. The information for the argument comes from the system architecture slices and formal analysis of the code. Text highlighted in red is provided by the formal analysis and provided by the specified information about the component, while all texts in black are boiler-plate i.e., can be reused in new contexts or applications without being changed much from the original.</p> <p>Possible Pitfalls</p> <ul style="list-style-type: none"> • The further argument of GFC1 relies on the complete list of the relevant external certification assumptions. • The further argument of GC1 relies on the fact that there is no missing signal relevant to the component.
Examples	As shown in Section 6.3 Figure 6.4
Known Uses	
Related Pattern	<p><i>System-Level Requirements to Component-Level Requirements</i> – This pattern is used to derive GC1</p> <p>This pattern forms part of a hierarchical safety cases, which includes the following patterns:</p> <ul style="list-style-type: none"> ▪ System-Level Requirements to Component-Level Requirements ▪ Sufficiency of Safety Condition (Chapter 4, Section 4.3.3 with some minor modifications) ▪ Explaining the Safety Notion (Chapter 4, Section 4.3.1 with some minor modifications)

Chapter 7

Proof-Oriented Safety Cases

Chapter 7 presents an approach to develop safety cases that correspond to formal proofs found by automated theorem provers. Their goal is to reveal the proof’s underlying argumentation structures and top-level assumptions.

7.1 Introduction

Demonstrating the correctness of large and complex software-intensive systems requires marshalling large amounts of diverse information, including models, code, specifications, mathematical equations and formulas, and tables of engineering constants. Tools supported by automated analyzers can be used to produce a traceable safety argument [48] that shows in particular where the code, verification artifacts, and the argument itself depend on any external assumptions.

Many tools commonly applied to ensure software safety rely on black-box techniques such as static analysis [18] or model checking [37] that produce only opaque claims about the safety of the software but not enough evidence to justify their claim. They can thus not provide any further insights or arguments. In contrast, in formal software safety certification [45], as in other formal software development methods [27, 28], formal proofs can in principle be used as evidence. Such proofs use mathematical and logical reasoning to show that the software satisfies certain requirements, as discussed in the previous three chapters. However, in practice there are reservations about the use of formal proofs as evidence (let alone arguments) in safety-critical applications. Concerns that the proofs may be based on assumptions that are not valid, or may contain steps that are not justified, can undermine the reasoning used to support the assurance claim. Moreover, these proofs are typically constructed automatically by automated theorem provers (ATPs) based on machine-oriented calculi such as resolution [134] which are often too complex and too difficult to understand by engineers, because the formalisms

spell out too many low-level details. In this chapter, we address these issues by systematically constructing safety cases that correspond to formal proofs found by ATPs and explicitly call out the use of external assumptions. The safety cases highlight the argument structure underlying the proof construction, and help showing how the truth of the theorem follows from the different assertions and subgoals.

The approach presented here combines abstraction and visualization to reveal and present the proofs underlying argumentation structure and top-level assumptions. We work with natural deduction (ND) style proofs, which are goal-directed (i.e., start with original theorem to be proven, and decompose it into subgoals) and closer to human reasoning than resolution proofs, and we show how the approach can be applied to the proofs found by the Muscadet ATP [118]. However, the approach is indirectly applicable to more powerful resolution provers as well, because resolution proofs can in principle be converted to ND-proofs [25, 79]. We explain how to construct the safety cases by covering the ND proof tree with corresponding safety case fragments. The argument is built in the same top-down way as the proof: it starts with the original theorem to be proved as the top goal and follows the deductive reasoning into subgoals, using the applied inference rules as strategies to derive the goals. However, we abstract away the ATPs book-keeping steps, which reduce the size of the constructed safety cases. The safety cases thus provide a “structured reading guide” for the proofs that allows users to understand the claims without having to understand all the technical details of the formal proof machinery.

7.2 Converting Natural Deduction Proofs into Safety Cases

Natural deduction [83] systems consist of a collection of proof rules that manipulate logical formulas and transform premises into conclusions. A conjecture is proven from a set of assumptions if a repeated application of the rules can establish it as conclusion. Here, we focus on some of the basic rules; a full exposition of the ND calculus can be found in the literature [83]

7.2.1 Conversion Process

Natural deduction proofs are simply trees that start with the conjecture to be proven as root, and have given axioms or assumed hypotheses at each leaf. Each non-leaf node is recursively justified by the proofs that start with its children as new conjectures. The edges between a node and all of its children correspond to the inference rule applied in this proof step. The proof tree structure is thus a representation of the underlying argumentation structure. We can use this interpretation to present the proofs as safety cases [86], which are structured arguments as well and also represent linkage between

evidence (i.e., the deductive reasoning of the proofs from the assumptions to the derived conclusions) and claims (i.e., the original theorem to be proved). The general idea of the conversion from ND proofs to safety cases is thus fairly straightforward. We consider the conclusion as a goal to be met and the premise(s) as a subgoal(s); we further consider the applied inference rule as the strategy that shows how the conclusion is met. For each inference rule, we define a safety case template that represents the same argumentation. The underlying similarity of proofs and safety cases has already been indicated in [86] but as far as we know, this idea has never been fully explored or even been applied to machine-generated proofs.

The conversion we present here preserves the inferences and formulas of the original proof, but avoids overloading the constructed arguments with trivial proof steps. We identify repeated identical inferences that can be abstracted away in order to construct a more concise argument. We also specify which inferences are semantically related and thus can be grouped together. In the following we describe the safety case templates for the base rules of the calculus. We use the Goal Structuring Notation [86] to explicitly represent the logical flow of the proofs argumentation structure.

7.2.1.1 Conjunctions

The rules for conjunction introduction and elimination directly represent the intuitive interpretation of conjunctions: if A is true and B is true, then evidently $A \wedge B$ is true as well (\wedge -i), and if $A \wedge B$ is true, then both A and B must be as well (\wedge -e₁ resp. \wedge -e₂).

$$\frac{A \quad B}{A \wedge B} (\wedge\text{-i}) \qquad \frac{A \wedge B}{A} (\wedge\text{-e}_1) \quad \frac{A \wedge B}{B} (\wedge\text{-e}_2) \qquad (7.1)$$

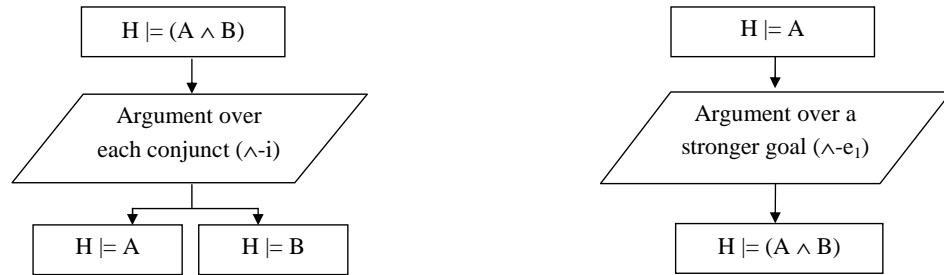


FIGURE 7.1: Safety Case Templates for \wedge -Rules

The hypotheses (H) that are available to show $A \wedge B$ true are also available to show A (resp. B) true as well. Similarly, in the case of \wedge -e₁ (resp. \wedge -e₂) the available hypotheses to show each conjunct true are also available to show $A \wedge B$ true.

The \wedge -rules can be directly converted into safety cases. In the case of \wedge -introduction, the

satisfaction of the conclusion (i.e., goal of the safety case) is implied by the satisfaction of the two premises (i.e., subgoals of the safety case), based on the strategy of the \wedge -introduction rule. For the \wedge -elimination rule, the strategy shows a logically stronger goal: we can conclude A (resp. B) if we have a proof of $A \wedge B$. Figure 7.1 shows the safety case templates for the conjunction rules.

7.2.1.2 Disjunctions

A disjunction can be introduced as long as one of the disjuncts is already established i.e., if A (resp. B) is true, then evidently $A \vee B$ is true as well. In the safety case, a goal $A \vee B$ is constructed, which is justified by the subgoal A (resp. B) via the strategy \vee - i_1 (resp. \vee - i_2). The hypotheses (H) that are available to show $A \vee B$ true are also available to show the subgoal A (resp. B) true.

In contrast, in disjunction elimination, we only know that $A \vee B$ holds, but not which of A or B is true, so that we need to reason by cases to draw any conclusion C from $A \vee B$, i.e., separately consider each of the two cases for the disjunction to be true. In the first case we thus assume A together with the available hypotheses and try to derive C , in the second case we assume B together with the available hypotheses and try to derive C . If both cases succeed, we can conclude C . The safety case fragment makes this argument explicit, and, in particular, explicitly justifies the use of the respective assumptions in the two cases. Figure 7.2 shows the safety case templates for the disjunction rules.

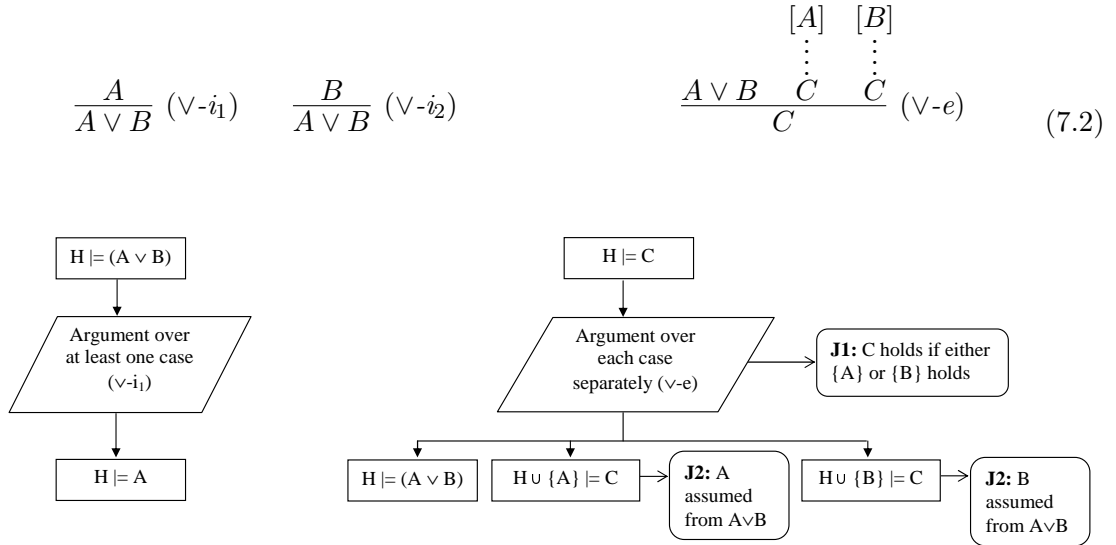


FIGURE 7.2: Safety Case Templates for \vee -Rules

7.2.1.3 Implications

The implication elimination follows the standard pattern but in the introduction rule, we again temporarily assume A as hypothesis together with the list of other available hypotheses, rather than deriving a proof for it. We then proceed to derive B , and *discharge* the hypothesis by the introduction of the implication. The hypothesis A can be used as given hypothesis in the proof of B , but the conclusion $A \Rightarrow B$ no longer depends on the hypothesis A after B has been proved. In the safety case template (see Figure 7.3), we use a justification to record the use of the hypothesis A , and thus to make sure that the introduced hypotheses are tracked properly.

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \Rightarrow B} (\Rightarrow -i) \qquad \frac{A \quad A \Rightarrow B}{B} (\Rightarrow -e) \quad (7.3)$$

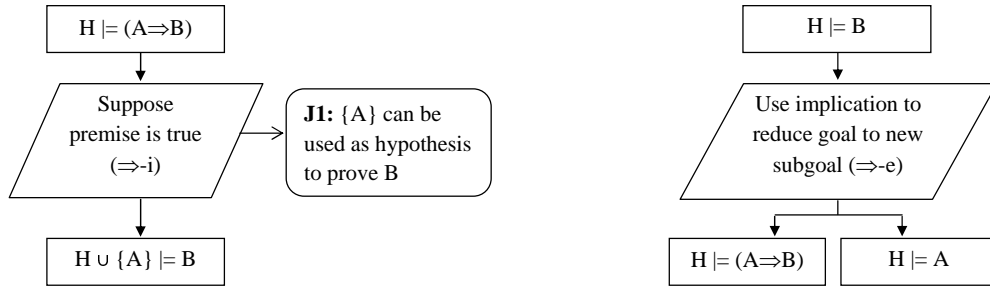
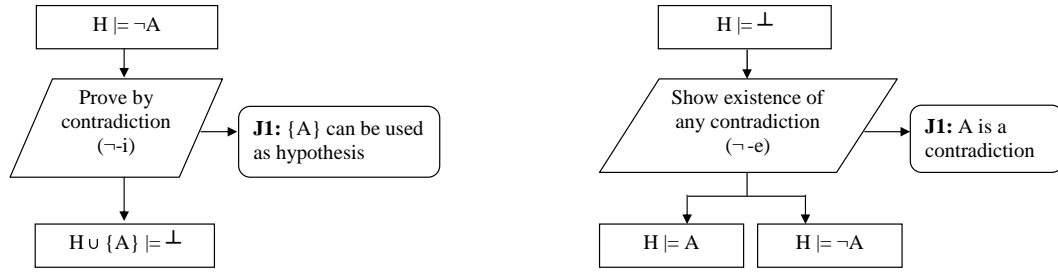


FIGURE 7.3: Safety Case Templates for \Rightarrow -Rules

7.2.1.4 Negations

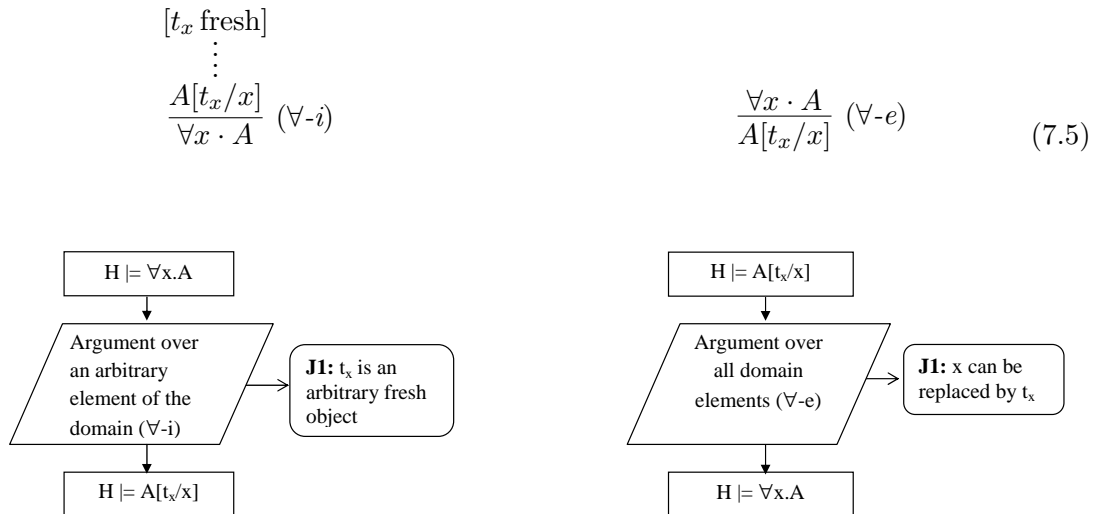
In natural deduction, the negation rules involve the notion of contradiction. The negation introduction is based on proof by contradiction i.e., to prove $\neg A$, we have to show that assuming A together with the hypotheses H leads to an inconsistent state. In the safety case template (see Figure 7.4), we use a justification to record the use of the hypothesis A which leads to an inconsistent state, i.e., \perp . In contrary, the negation elimination rule says the falsehood (\perp) follows from any contradiction of the premises A and $\neg A$ (as shown in Figure 7.4). The hypotheses (H) that are available to show A and $\neg A$ true are also available to show the falsehood to derive an inconsistent state as well.

$$\frac{\begin{array}{c} [A] \\ \vdots \\ \perp \end{array}}{\neg A} (\neg -i) \qquad \frac{A \quad \neg A}{\perp} (\neg -e) \quad (7.4)$$

FIGURE 7.4: Safety Case Templates for \neg -Rules

7.2.1.5 Universal quantifiers

The natural deduction calculus can also be used for proofs in predicate logic. The proof rules focus on the replacement of the bound variables with objects and vice versa. For example, in the elimination rule for universal quantifiers (\forall), we can conclude the validity of the formula for any chosen domain element t_x . In the introduction rule (\forall_i), however, we need to show the formula for an arbitrary but fresh object t_x , that is, a domain element which does not appear elsewhere in H , A , or the domain theory and assumptions. t_x is also called the eigenvariable of the rule. If we can derive a proof of A , where x is replaced by the object t_x , we can then discharge this assumption by introduction of the quantifier \forall . The safety case fragments (see Figure 7.5) record this replacement (i.e., the choice of eigenvariable) as justification. The hypotheses available for the subgoals in the \forall -rules are the same as those in the original goals.

FIGURE 7.5: Safety Case Templates for \forall -Rules

7.2.1.6 Existential quantifiers

In the existential introduction rule (see Equation 7.6), we can conclude the validity of the formula $\exists x \cdot A$ if we have found a witness of the formula A , i.e., can prove it for an arbitrary object t_x . In existential elimination, given that $\exists x \cdot A$ holds, we can conclude that C is true if we can derive a proof of C from A where x has been replaced by an arbitrary fresh object t_x , i.e., an eigenvariable. This follows the same structure as in the case of disjunctions. The safety case fragments (see Figure 7.6) again record these replacements as justification.

In the introduction rule, the hypotheses that are available to show $\exists x \cdot A$ true are available to show $A[t_x/x]$ true as well. Similarly, in the case of elimination rule, the available hypotheses to show C true in the conclusion are also available to show $\exists x \cdot A$, but in the premise, we also have $A[t_x/x]$ available to show C .

$$\frac{A[t_x/x]}{\exists x \cdot A} (\exists-i) \qquad \frac{\exists x \cdot A \quad \begin{array}{c} t_x A[t_x/x] \\ \vdots \\ C \end{array}}{C} (\exists-e) \quad (7.6)$$

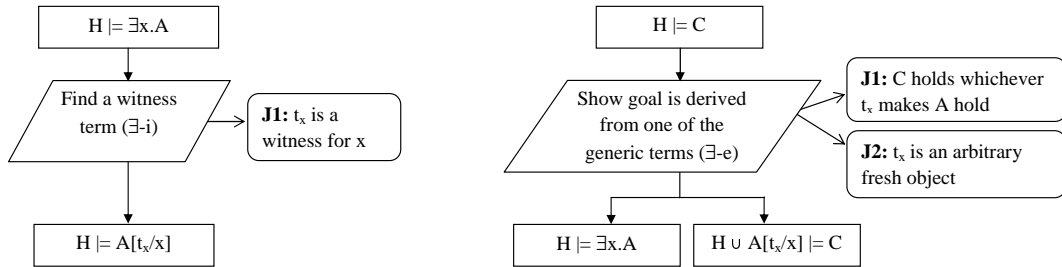


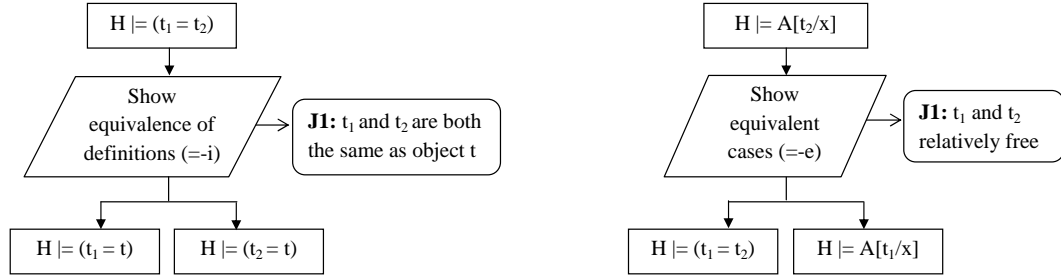
FIGURE 7.6: Safety Case Templates for \exists -Rules

7.2.1.7 Equalities

In Muscadet, equality is treated by special reasoning rules based on the axiomatic definition of equality. The three axioms of reflexivity, symmetry, and transitivity are combined into one special introduction rule that uses the existence of a common equal domain object to show the equalities. For example, when two terms t_1 and t_2 are equal to t , we can conclude that t_1 and t_2 are equal, as shown in the safety case fragments Figure 7.7.

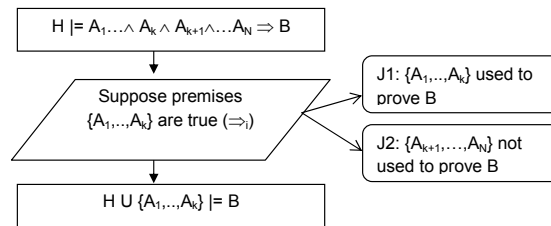
In the equality elimination rule ($=_e$), if we know the terms t_1 and t_2 are equal, we can replace any occurrences of t_1 in a true proposition and obtain another true proposition with t_2 . However both t_1 and t_2 have to be relatively free for A , i.e., contain no occurrence of variables bound by A . In the safety case fragments (see Figure 7.7), we indicate this condition on t_1 and t_2 in A as justification.

$$\frac{}{t = t} (refl) \quad \frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3} (trans) \quad \frac{t_1 = t_2}{t_2 = t_1} (symm) \quad \frac{t_1 = t_2 \quad A[t_1/x]}{A[t_2/x]} (= -e) \quad (7.7)$$

FIGURE 7.7: Safety Case Templates for $=$ -Rules

7.3 Hypothesis Handling

An automated prover typically treats the domain theory D and the certification assumptions A as premises and tries to derive $D \wedge A \wedge P \Rightarrow C$ (where P are the original premises and C is the conclusion) from an empty set of hypotheses. As the proof tree grows, these premises will be turned into hypotheses, using the \Rightarrow -introduction rule (see Figure 7.3). However, not all premises will actually be used as hypotheses in the proof, and the safety case should highlight those that are actually used. This is particularly important for the certification assumptions. We can achieve this by modifying the template for the \Rightarrow -introduction (see Figure 7.8). We distinguish between the hypotheses that are actually used in the proof of the conclusion (denoted by A_1, \dots, A_k) and those that are vacuously discharged by the \Rightarrow -introduction (denoted by $A_1, \dots, A_k, A_{k+1}, \dots, A_n$). We can use two different justifications to mark this distinction. Note that this is only a simplification of the presentation and does not change the structure of the underlying proof, nor the validity of the original goal. It is thus different from using a relevant implication [24] under which $A \Rightarrow B$ is only valid if the hypothesis A is actually used.

FIGURE 7.8: Hypothesis Handling in \Rightarrow -Rules

In order to minimize the number of hypotheses tracked by the safety case, we need to analyze the proof tree from the leaves up, and propagate the used hypotheses towards the root. By revealing only the used hypotheses as assumptions, the validity of their use in deriving the proof can be checked more easily. In our work, we also highlight the use of the external certification assumptions that have been formulated in isolation by the safety engineer. For example in Figure 7.9, the use of the hypothesis $has_unit(0.7, ang_vel)$, meaning that this particular floating point number represents an angular velocity, which has been specified as external assumption, is tracked properly in the safety case. Muscadet turns this external hypothesis (given in a file containing the certification assumptions) into an axiom, and the use of this axiom can be justified and tracked properly, and can be checked easily.

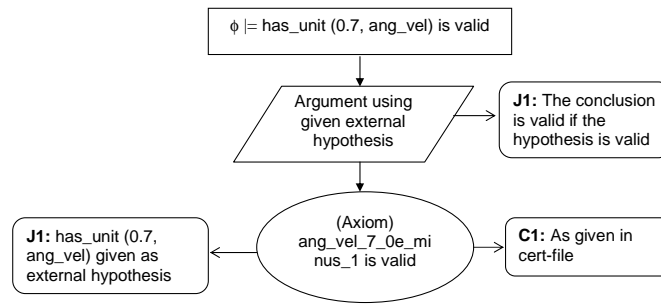


FIGURE 7.9: External Hypothesis

7.4 Application to Muscadet Prover

We illustrate our approach by converting proofs by the Muscadet [118] theorem prover during the formal program verification of the initialization safety of a component of an attitude control system. Muscadet is based on natural deduction, but to improve performance, it implements a variety of derived rules in addition to the basic rules of the calculus. This includes rules for dedicated equality handling, as well as rules that the system builds from the definitions and lemmas, and that correspond to the application of the given definitions and lemmas.

The Muscadet prover is based on an inference engine, which interprets and executes the rules of the calculus for the theorem to be proved. The inference engine manages an interval state for each theorem or sub-theorem to be proved which consists of:

- objects that were created,
- hypotheses that are used,
- conclusion to be proved,
- active rules that are used,

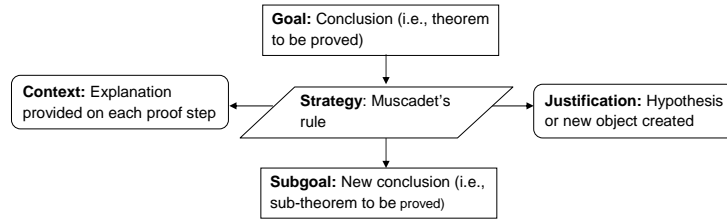


FIGURE 7.10: Generic Safety Case Template to Handle Muscadet Book Keeping Rules

- explanations of the strategy, and
- new conclusions or sub-theorems.

Muscadet’s proof strategy starts with the initial statement of the theorem to be proved. Each rule may add new hypotheses, modify the conclusion, create objects, create sub-theorems or build new rules which are local for (sub)theorems. We defined a dedicated safety case template for the Muscadet “book-keeping” rules (e.g., the elimination of function applications known as the elifun-rule and initialization of the theorem known as the ini-rule). The template is customizable and can be applied to other Muscadet rules as well (i.e., these rules are represented by a generic strategy node). Figure 7.10 shows the generic template to represent the book-keeping rules.

We then use the customizable Muscadet template together with ND safety case templates as defined in Section 7.2.1 to transform and represent the resulting proof found by the Muscadet prover into a safety case. For a first example, let us consider the proof of the transitivity of inclusion (as given in [118]),

$$\forall A \forall B \forall C (A \subset B \wedge B \subset C \Rightarrow A \subset C)$$

In order to prove this theorem, Muscadet creates three objects a, b and c by applying the \forall -elimination rule

Rule \forall : if the conclusion is $\forall X p(X)$, then create a new object x and the new conclusion is $p(x)$

three times and thus, the new conclusion is $a \subset b \wedge b \subset c \Rightarrow a \subset c$. Then the \Rightarrow -introduction rule

Rule \Rightarrow : if the conclusion is $H \rightarrow C$, then add the hypothesis H and the new conclusion is C

replaces the conclusion by $a \subset c$ and adds two hypotheses $a \subset b$ and $b \subset c$.

Figure 7.11 shows the corresponding safety case after these steps. The theorem to be proved (i.e., the transitivity of inclusion) is the top goal of the safety case, the new conclusion of each proof step occurs as a sub-goal of the argument and the applied inference rule as the strategy. The new object created in each proof step is considered as a justification to the strategy.

Returning to our certification example, we construct a safety case for the proof of a VC from showing the initialization-before-use safety property of some code generated by the AutoFilter system (see Figure 7.12). The same conversion process as described for proving the transitivity of inclusion above was applied in this example. In order to fit the safety case to the page size, we use labels to represent the actual theorem and hypotheses. For example, ϕ_1 indicates the first conclusion or theorem and h_1 indicates the first hypothesis. We preserve the name of the applied inference rules as used by Muscadet. In Figure 7.12, the theorem to be proved (labelled as ϕ with empty set of hypothesis) is:

$$\begin{aligned}
& (0 \leq \text{pv5}) \wedge (\text{pv5} \leq 0) \wedge (\text{pv5} \leq 998) \wedge (\text{pv5} > 0) \wedge \forall [A,B] : ((0 \leq A) \wedge (0 \leq B) \wedge (A \leq 5) \wedge (B \leq 0)) \\
& \Rightarrow \text{a_select3}(\text{xhatmin_ds1}, A, B) :: \text{init} \wedge \forall [C,D] : ((0 \leq C) \wedge (0 \leq D) \wedge (C \leq 5) \wedge (D \leq 0)) \\
& \Rightarrow \text{a_select3}(\text{xhatmin_ds1}, C, D) :: \text{init} \wedge \forall [E, F] : ((0 \leq E) \wedge (0 \leq F) \wedge (E \leq 2) \wedge (F \leq 2)) \\
& \Rightarrow \text{a_select3}(\text{r_ds1}, E, F) :: \text{init} \wedge \forall [G,H] : ((0 \leq G) \wedge (0 \leq H) \wedge (G \leq 5) \wedge (H \leq 5)) \\
& \Rightarrow \text{a_select3}(\text{q_ds1}, G, H) :: \text{init} \wedge \forall [I, J] : ((0 \leq I) \wedge (0 \leq J) \wedge (I \leq 5) \wedge (J \leq 5)) \\
& \Rightarrow \text{a_select3}(\text{pminus_ds1}, I, J) :: \text{init} \wedge \forall [K,L] : ((0 \leq K) \wedge (0 \leq L) \wedge (K \leq 5) \wedge (L \leq 5)) \\
& \Rightarrow \text{a_select3}(\text{pminus_ds1}, K, L) :: \text{init} \wedge \forall [M,N] : ((0 \leq M) \wedge (0 \leq N) \wedge (M \leq 5) \wedge (N \leq 5)) \\
& \Rightarrow \text{a_select3}(\text{phi_ds1}, M, N) :: \text{init} \wedge \forall [O] : ((0 \leq O) \wedge (O \leq 5)) \wedge \forall [P] : ((0 \leq P) \wedge (P \leq 5)) \\
& \Rightarrow \text{a_select3}(\text{id_ds1}, O, P) :: \text{init} \wedge \forall [Q,R] : ((0 \leq Q) \wedge (0 \leq R) \wedge (Q \leq 2) \wedge (R \leq 5)) \\
& \Rightarrow \text{a_select3}(\text{h_ds1}, Q, R) :: \text{init} \wedge ((\text{pv5} \leq 0) \wedge (\text{pv5} > 0)) \Rightarrow \forall [S,T] : ((0 \leq S) \wedge (0 \leq T) \wedge (S \leq 5) \wedge (T \leq 0)) \\
& \Rightarrow \text{a_select3}(\text{xhatmin_ds1}, S, T) :: \text{init} \wedge ((\text{pv5} \leq 0)) \Rightarrow \forall [U, V] : ((0 \leq U) \wedge (0 \leq V) \wedge (U \leq 5) \wedge (V \leq 0)) \\
& \Rightarrow \text{a_select3}(\text{xhatmin_ds1}, U, V) :: \text{init} \wedge ((\text{pv5} > 0)) \Rightarrow \forall [W,X] : ((0 \leq W) \wedge (0 \leq X) \wedge (W \leq 5) \wedge (X \leq 0)) \\
& \Rightarrow \text{a_select3}(\text{xhatmin_ds1}, W, X) :: \text{init} \Rightarrow \text{a_select3}(\text{xhatmin_ds1}, 3, 0) :: \text{init}
\end{aligned}$$

The proof consists of seventeen rule applications. We add another strategy (i.e., show hypotheses are valid) to communicate the flow of the argument. To prove that the theorem holds, the theorem is decomposed into sub-theorems (or sub-goals), based on the natural deduction rules. In the safety case, contexts explain additional information for the proof strategy such as the proofs are natural deduction proofs and found by the Muscadet prover. Table 7.1 shows the details of each label as used in Figure 7.12. Here *leq* and *gt* represent \leq and \geq , respectively, and *a_select3* represents array indexing, with its first argument the array and the other two the index, *::* represents the equality com-

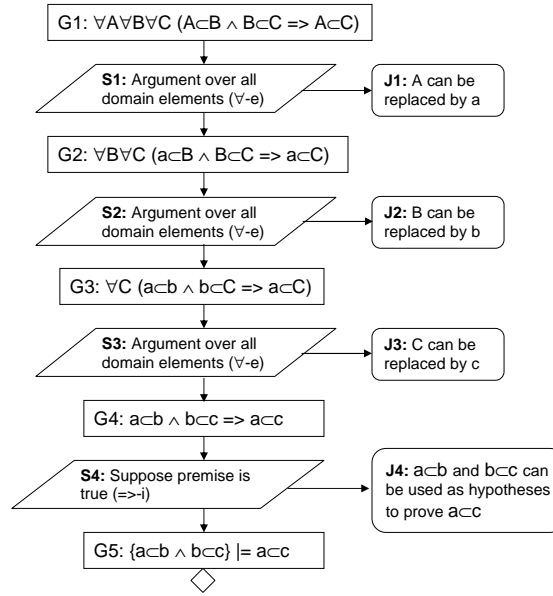


FIGURE 7.11: Example of Muscadet ND Proof Safety Case

combined with function application, initialization and elifun represent internal Muscadet's book-keeping rules for initialization of the theorem and elimination of the functional symbols of the theorem. A link from G19 to G7 indicates the validity of G19 can be shown by the argument of G7.

7.5 Proof Abstraction

The example in Figure 7.12 shows that directly converting the Muscadet-proofs into safety cases is unfeasible in most practical cases because the proofs contain too many elementary and book-keeping steps. It is thus necessary to abstract the proof. Here, we can apply different approaches. Several Muscadet book keeping rules such as elifun (i.e., elimination of functional symbols of the conclusion) and return_proof (i.e., the theorem has been proved) do not affect the underlying proof reasoning strategy and are not central to the overall argumentation structure, so that they can be removed from the safety case (see Figure 7.15). Similarly, we can collapse sequences of identical rules into a single node and represent them in the proof argument as a single strategy, without losing any insights into the structure and readability of the proof. We can label the consecutive occurrences of these rules in the proof safety case by $(*N)$ where N is an integer that indicates N rule occurrences, as shown in Figure 7.13.

In general, however, we try to restructure the resulting proof presentation to help in emphasizing the essential proof steps. In particular, we can group sub-proofs that apply only axioms and lemmas from certain obvious parts of the domain theory (e.g., ground arithmetic or partial order reasoning) and represent them as a single strategy application.

TABLE 7.1: Detail of Labels as used in Figure 7.12

Label	Theorem or Hypothesis
$\phi 1, \phi 2$	$(0 \leq pv5) \wedge (pv5 \leq 0) \wedge (pv5 \leq 998) \wedge (pv5 > 0)$ $\wedge \forall [A,B] : ((0 \leq A) \wedge (0 \leq B) \wedge (A \leq 5) \wedge (B \leq 0)) \Rightarrow a_select3(xhatmin_ds1, A,B) :: init$ $\wedge \forall [C,D] : ((0 \leq C) \wedge (0 \leq D) \wedge (C \leq 5) \wedge (D \leq 0)) \Rightarrow a_select3(xhatmin_ds1, C,D) :: init$ $\wedge \forall [E, F] : ((0 \leq E) \wedge (0 \leq F) \wedge (E \leq 2) \wedge (F \leq 2)) \Rightarrow a_select3(r_ds1, E, F) :: init$ $\wedge \forall [G,H] : ((0 \leq G) \wedge (0 \leq H) \wedge (G \leq 5) \wedge (H \leq 5)) \Rightarrow a_select3(q_ds1, G,H) :: init$ $\wedge \forall [I, J] : ((0 \leq I) \wedge (0 \leq J) \wedge (I \leq 5) \wedge (J \leq 5)) \Rightarrow a_select3(pminus_ds1, I, J) :: init$ $\wedge \forall [K,L] : ((0 \leq K) \wedge (0 \leq L) \wedge (K \leq 5) \wedge (L \leq 5)) \Rightarrow a_select3(pminus_ds1, K,L) :: init$ $\wedge \forall [M,N] : ((0 \leq M) \wedge (0 \leq N) \wedge (M \leq 5) \wedge (N \leq 5)) \Rightarrow a_select3(phi_ds1, M,N) :: init$ $\wedge \forall [O] : ((0 \leq O) \wedge (O \leq 5)) \vee [P] : ((0 \leq P) \wedge (P \leq 5)) \Rightarrow a_select3(id_ds1, O, P) :: init$ $\wedge \forall [Q,R] : ((0 \leq Q) \wedge (0 \leq R) \wedge (Q \leq 2) \wedge (R \leq 5)) \Rightarrow a_select3(h_ds1, Q,R) :: init$ $\wedge ((pv5 \leq 0) \wedge (pv5 > 0)) \Rightarrow \forall [S,T] : ((0 \leq S) \wedge (0 \leq T) \wedge (S \leq 5) \wedge (T \leq 0)) \Rightarrow a_select3(xhatmin_ds1, S, T) :: init$ $\wedge ((pv5 \leq 0) \wedge (pv5 > 0)) \Rightarrow \forall [U, V] : ((0 \leq U) \wedge (0 \leq V) \wedge (U \leq 5) \wedge (V \leq 0)) \Rightarrow a_select3(xhatmin_ds1, U, V) :: init$ $\wedge ((pv5 > 0) \wedge (pv5 \leq 0)) \Rightarrow \forall [W,X] : ((0 \leq W) \wedge (0 \leq X) \wedge (W \leq 5) \wedge (X \leq 0)) \Rightarrow a_select3(xhatmin_ds1, W,X) :: init$ $\Rightarrow a_select3(xhatmin_ds1, 3, 0) :: init$
$\phi 3$	$a_select3(xhatmin_ds1, 3, 0) :: init$
$\phi 4$	$only(a_select3(xhatmin_ds1_filter_init, 3, 0) :: A, A=init)$
$\phi 5$	$z1=init$
$\phi 6$	$init=init$
$h1$	$leq(0, pv5)$
$h2$	$leq(pv5, 0)$
$h3$	$leq(pv5, 998)$
$h4$	$gt(pv5, 0)$
$h5$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 5) \wedge leq(B, 0) \Rightarrow a_select3(xhatmin_ds1_filter_init, A, B)::init$
$h6$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 5) \wedge leq(B, 0) \Rightarrow a_select3(xhatmin_ds1_filter_init, A, B)::init$
$h7$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 2) \wedge leq(B, 2) \Rightarrow a_select3(r_ds1_filter_init, A, B)::init$
$h8$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 5) \wedge leq(B, 5) \Rightarrow a_select3(q_ds1_filter_init, A, B)::init$
$h9$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 5) \wedge leq(B, 5) \Rightarrow a_select3(pminus_ds1_filter_init, A, B)::init$
$h10$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 5) \wedge leq(B, 5) \Rightarrow a_select3(pminus_ds1_filter_init, A, B)::init$
$h11$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 5) \wedge leq(B, 5) \Rightarrow a_select3(phi_ds1_filter_init, A, B)::init$
$h12$	$![A]: leq(0,A) \wedge leq(A, 5) \Rightarrow ![B]: leq(B, 5) \wedge leq(B, 5) \Rightarrow a_select3(id_ds1_filter_init, A, B)::init$
$h13$	$![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A, 2) \wedge leq(B, 5) \Rightarrow a_select3(h_ds1_filter_init, A, B)::init$
$h14$	$leq(pv5,0) \wedge gt(pv5,0) \Rightarrow ![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A,5) \wedge leq(B,0)$ $\Rightarrow a_select3(xhatmin_ds1_filter_init, A, B)::init$
$h15$	$leq(pv5,0) \Rightarrow ![A,B]: leq(0,A) \wedge leq(0,B) \wedge leq(A,5) \wedge leq(B,0) \Rightarrow a_select3(xhatmin_ds1_filter_init, A, B)::init$
$h16$	$gt(pv5, 0) \Rightarrow ![A, B]: (leq(0, A) \wedge leq(0, B) \wedge leq(A, 5) \wedge leq(B, 0)) \Rightarrow a_select3(xhatmin_ds1_filter_init, A, B)::init$
$h17$	$a_select3(xhatmin_ds1_filter_init, 3, 0)::z1$
$h18$	$leq(0, 0)$
$h19$	$gt(5, 4)$
$h20$	$leq(4, 5)$
$h21$	$gt(3, 0)$
$h22$	$leq(0, 3)$
$h23$	$gt(4, 3)$
$h24$	$leq(3, 4)$
$h25$	$leq(3, 5)$
$h26$	$a_select3(xhatmin_ds1_filter_init, 3, 0)::init$
$h27$	$z1=init$

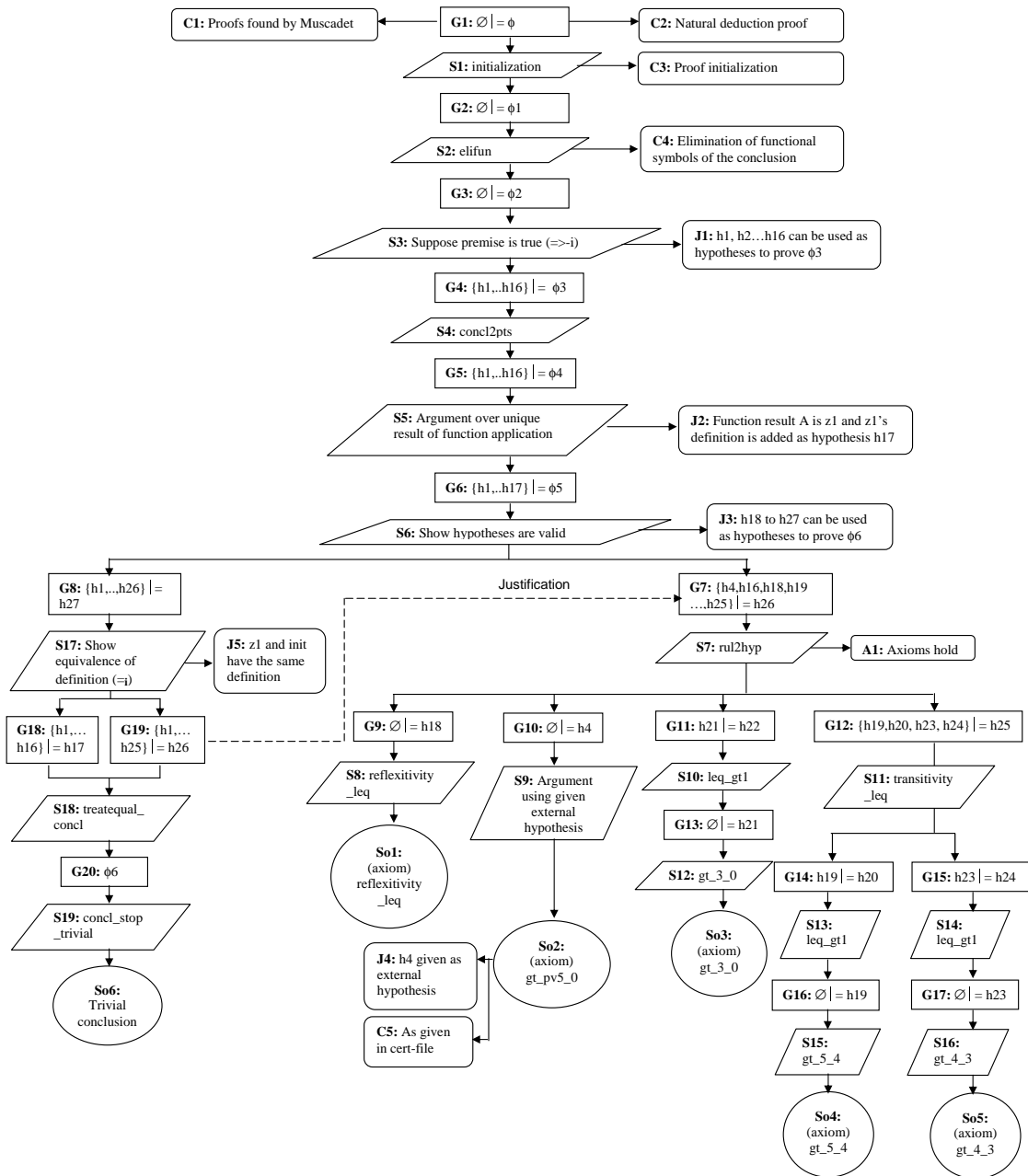


FIGURE 7.12: ND Proof Safety Case of Initialization-Before-Use

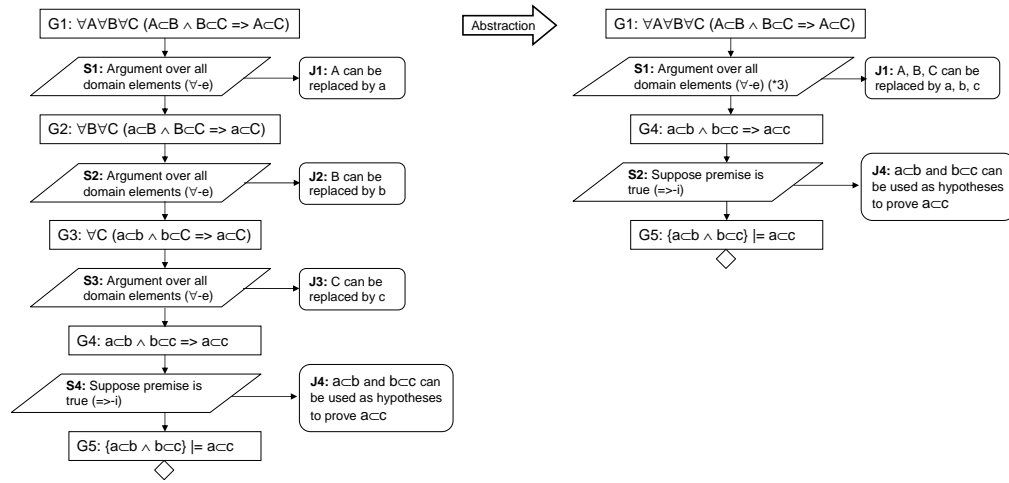
FIGURE 7.13: Abstraction of Consecutive Rules (*concl_only*)

Figure 7.14 shows an example of this. Here, the first abstraction step collapses the sequences rooted in G14 and G15, noting the lemmas which had been used as strategies as justifications, but keeping the branching that is typical for the transitivity. A second step then abstracts this away as well. Figure 7.15 shows the resulting safety case of the initialization-before-use proof for Figure 7.12 after abstraction.

The purpose of the abstraction is to reduce size and complexity of the overall proof presentation and to make the constructed safety cases more readable. We represent the deductive reasoning with a collapsible structured proof. We generalized the commonality of the low-level inferences and grouped them together. We also have made a decision as to how far to take the proof details. In our work, we classify the proof tiles into two categories, obvious and essential; and derive the safety case from the essential tiles. Obvious rules include Muscadet book keeping rules such as the *elifun* and *return_proof* rules. These rules can be abstracted away (i.e., the strategies can be removed in the safety case) if the goals can be directly achieved and understood without affecting understandability of the presentation of the underlying arguments. Essential rules are rules that should remain in the proof argument as they are essential in showing the correctness of the underlying the proofs. So far, we decide manually which rules are essential by looking at the overall proof presentation, and restructure the argument to make it more simplified and readable. However, we still keep the actual proofs argument for validity checking purposes.

7.6 Conclusions

Greater confidence can be placed in the assurance claims provided by proofs if the rationale behind the validity of the proofs can be shown. The probability of a claim, which has been shown by a formal proof, being false, is very low, when the assumptions

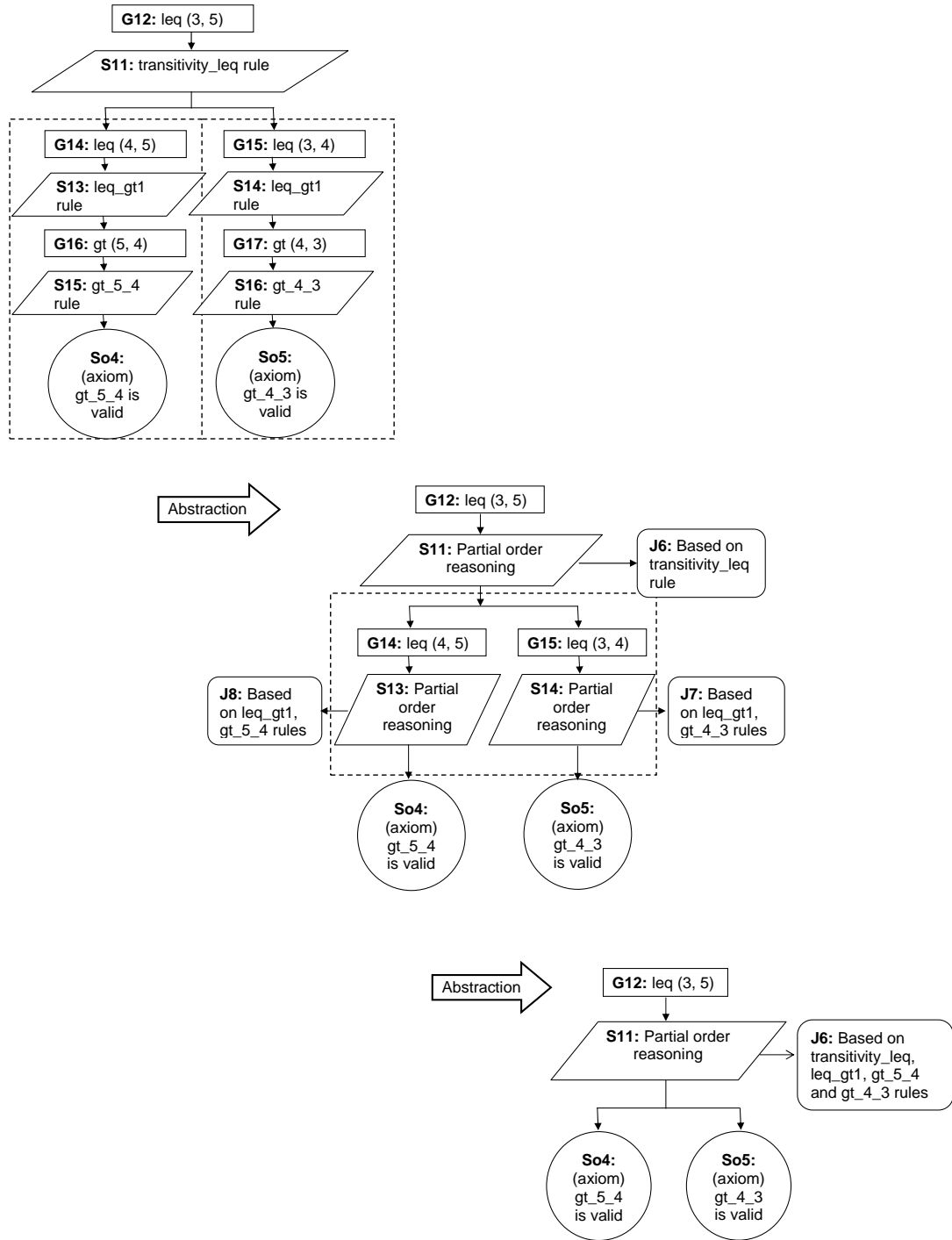


FIGURE 7.14: Abstraction of Partial Order Reasoning Rules

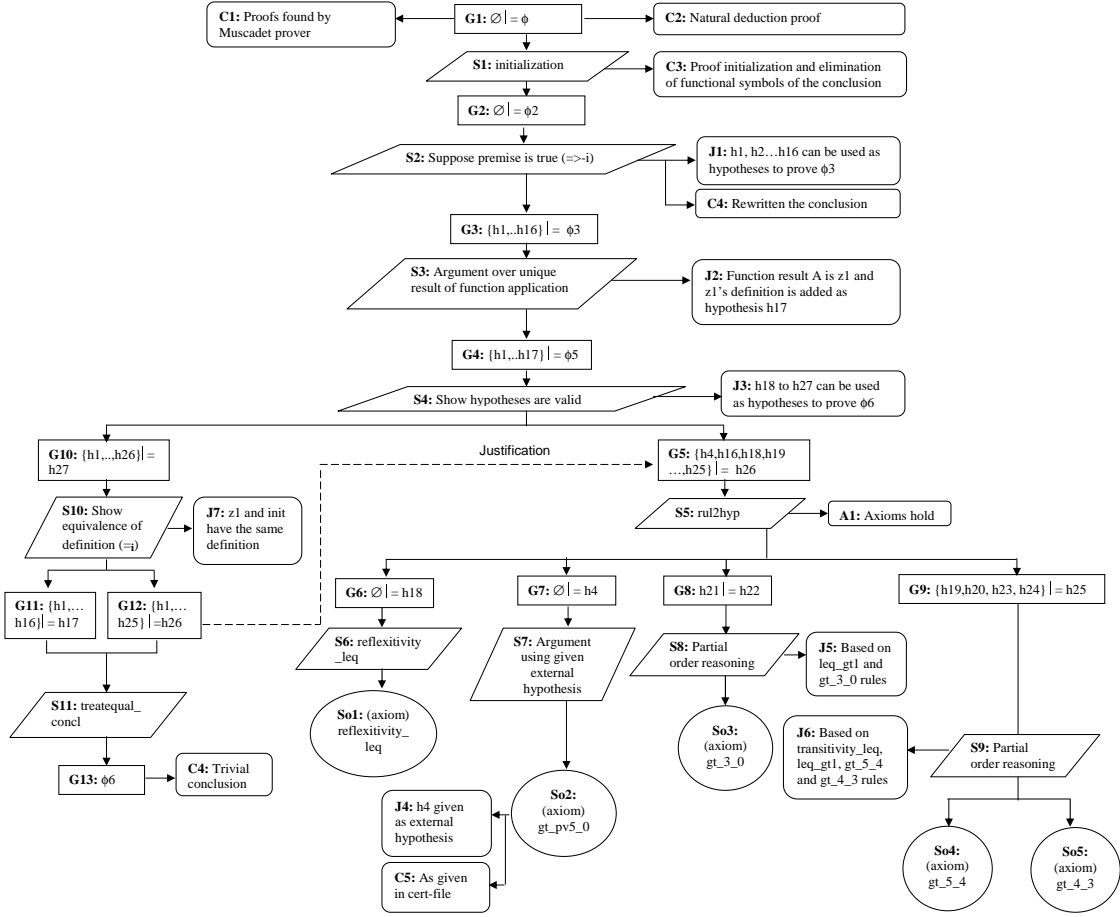


FIGURE 7.15: After Abstraction: Initialization-Before Use Proofs

and evidence are valid [100]. However, there is a non-zero probability that these are not in fact valid.

Therefore, in this chapter, we have described an approach to derive safety cases from the software safety proofs found by ATPs. The resulting safety cases serve as a traceable argument that show the validity of the proof. They also highlight and properly track hypotheses that are actually used in deriving the proof, and thus reveal where the proofs depend on top-level assumptions. Hence, assurance is no longer implied by the trust in the ATP or the proofs but follows from an explicitly constructed argument for the proofs. We illustrate our approach on natural deduction proofs found by the Muscadet prover [118]. However, a straightforward conversion of these proofs into safety cases is far from satisfactory as they typically contain too many details. A careful use of abstraction is needed [51] to reduce size and complexity of the overall proof presentation and to make the constructed safety cases more readable. Therefore, we have also introduced a proof abstraction technique to simplify the resulting safety cases. We believe the abstracted safety case is sufficient to make clear the argumentations underlying the proofs. We hope the same safety cases derivation can also be applied to other natural deduction theorem provers as well.

Chapter 8

Heterogenous Safety Cases

Chapter 8 focuses on merging the different safety cases described in the previous chapters into a single integrated safety case. It also shows how to add additional verification and validation information from heterogenous sources into that safety case.

8.1 Introduction

In the previous chapters, we have described an approach to construct different styles of safety cases (i.e., property-oriented, requirement-oriented, architecture-oriented and proof-oriented safety cases) from a formal analysis, based on automated theorem proving, of the automatically generated code. However, the main focus of the argument in each safety case is slightly different in terms of the safety aspect considered. For example, the property-oriented safety cases are focused on ensuring safety of the program wrt. a given safety property (such as initialization-before-use safety), while the architecture-oriented safety cases are concentrated on arguing along the hierarchical structure of the system. Therefore, in this chapter, we describe an approach to combine these different safety cases as well as diverse types of information from other verification and validation activities such as testing into a single integrated or *heterogenous* safety case, in order to provide comprehensive assurance of the program safety. This heterogenous safety case integrates additional information into the existing core argument structure of the previous safety cases representing the purely formal reasoning on the safety of the program. This additional verification and validation information represents background knowledge that cannot be produced directly by the formal verification phase (e.g., a justification that the formalizations of the properties that are verified correctly encode the requirements, or links to applicable standards and project documentation), and which thus needs to be specified in the form of contexts, assumptions, justifications, and constraints in the safety case. It can also represent additional forms of evidence which can be derived from other verification activities, such as testing. This information needs

to be spliced into the core argument structure of the heterogenous safety case at the appropriate locations to provide a traceable and comprehensive safety argument.

8.2 Constructing Heterogenous Safety Cases

In this section, we describe how to integrate the property-oriented, requirement-oriented, architecture-oriented and proof-oriented safety cases into a single integrated safety case, i.e., the heterogenous safety case. As described above, each of these safety cases focuses on arguing different safety aspects of the software such as safety properties, safety requirements, proofs, and system architecture. However, to achieve a comprehensive program verification argument, an integration of these different arguments should be described in a single view to strengthen the assurance provided. Moreover, the integration can also highlight the relation between the individual arguments and thus reduces repetition in the arguments. The main purpose of the heterogenous safety case is to provide a minimal but consistent safety case representing a combination of different categories of information.

Here, we also describe how we can combine the diverse types of information from auxiliary verification and validation sources (e.g., verification tool information or testing results) and splice them into the heterogenous safety case. However, we keep the tiered structure, and describe the overall heterogenous safety argument in its four different tiers, to help in showing the high-level structure of the argument. We provide a simplified overview of the safety case and describe it in a context of template instead of a full pattern, concentrating on its generic structure. As before, all texts highlighted in red in the heterogenous safety case need to be instantiated; the concrete text can be derived from the certification information, generator, software model and from the specified certification information such as requirement documents and testing results. All texts in black are boiler-plate which can be reused as is. Nodes labelled with triangle (\triangle) or with diamond (\diamond) need to be further instantiated or developed. Links with (\circ) describe the nodes that are optional and links with (\bullet) describe nodes that can be solved by zero or more instances.

8.2.1 Tier I of Heterogenous Safety Case: Explaining the Safety Notion

Tier I of the heterogenous safety case (see Figure 8.1) is adapted from the pattern “Explaining the Safety Notion” described on page 65 but with some minor modifications as pointed out here. In the previous chapters, the main focus of our argument was on assuring safety of the program wrt. the safety properties and the safety requirements respectively. Therefore, in the top level of the combined safety case, we explicitly describe

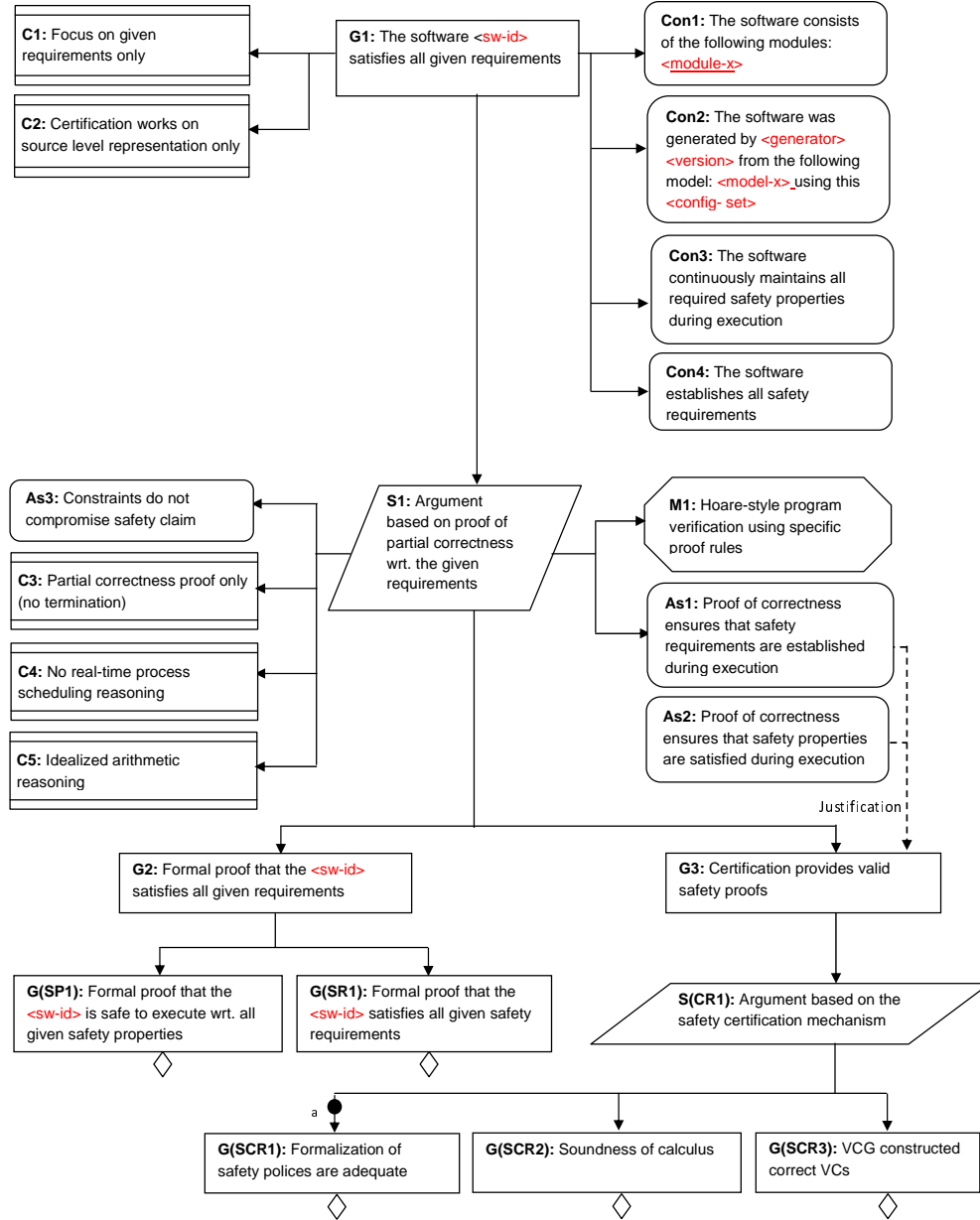


FIGURE 8.1: Tier I of the Heterogenous Safety Case: Explaining the Safety Notion

the context of the safety verification, i.e., define the meaning of the text “all given requirements” in the root goal (G1). Hence, the contexts Con3 and Con4 explicitly describe that the focus of the argument is on assuring software safety as specified by two categories of requirements, i.e., safety properties and safety requirements. Further to this, limitations to which extent the verification strategy can show the software safety are made explicit in form of the constraints C1 and C2. Con1 and Con2 describe the provenance of the software under certification.

The “proof of partial correctness” strategy, which is applied here is predicated on two assumptions, first, that the proof of correctness ensures that the safety requirements are

established during execution and second, that it ensures that the safety properties are satisfied during execution. Both of these assumptions can be justified by the goal (G3). The model underlying the argument (i.e., Hoare-style verification) is made explicit, as are the limitations of this model (C3-C5). An explicit assumption is made that these limitations do not compromise the safety claims; arguing the validity of this assumption is outside the scope of the work described here.

The strategy (S1) is then reduced to two subgoals, (G2) and (G3). The strategy's first subgoal (G2) is the “glue” of the overall argument is further elaborated into two subgoals, showing that the program is safe wrt. all given safety properties and safe wrt. all given safety requirements. The further argument structure of each subgoal is described in detail in tier II of the heterogenous safety case. The strategy's second subgoal is to show that the certification provides valid safety proofs which then spawns three subgoals, showing first, that G(SCR1): the formalization of the safety policy is adequate (which needs to be repeated for each of the safety properties considered), second, that G(SCR2): the calculus is sound to represent the safety of the requirements, and third, that G(SCR3): the VCG provides correct implementation of the VCs. However, these three subgoals are not elaborated further in this safety case but lead to the complementary safety case for the safety logic which is left for future work.

8.2.2 Tier II of Heterogenous Safety Case: Arguing over Safety of Program

The key argument strategy at tier II of the heterogenous safety case is to argue software safety to safety wrt. all given requirements. In order to show this, the argument is decomposed into two subgoals i.e., safety wrt. all given safety properties and safety wrt. all given safety requirements. Basically, the overall argument structure for showing the safety wrt. both styles of requirements follows the same structure as described in the “Safety of Program Variables” and “Formalization of Safety Requirement” safety case patterns, on page 70 and page 87, respectively.

Figure 8.2 shows the safety case pattern for arguing over all safety properties, which proceeds properly by property. For each individual property, the argument remains as described in the “Safety of Program Variables” pattern (see page 70), but some contexts (e.g., Con5, Con6 and Con7) and constraints (e.g., C6) have been moved from the original top tier to here, because there are specific to the (generic) notion of a safety property. The purpose of the safety case remains to reduce the program safety to the safety of all program variables, based on the fact that safety properties are defined on individual variables. The argument proceeds accordingly as in the pattern, until it reaches the argument on the establishment of the safety condition which is further described in tier III of the heterogenous safety case.

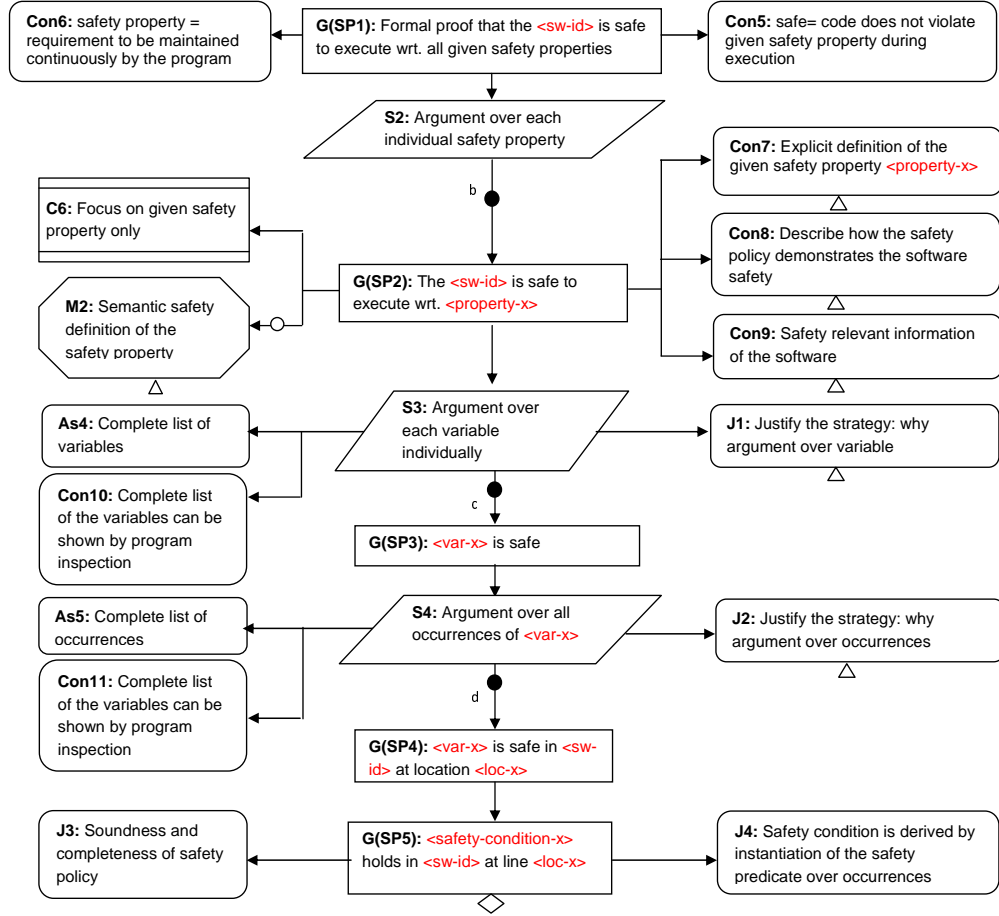


FIGURE 8.2: Tier II of the Heterogenous Safety Case: Arguing over Safety Property

Figure 8.3 shows the safety case pattern for safety requirements. The overall argument remains the same as described in the “Formalization of Safety Requirement” pattern, (see page 87), except that the branch argument over the formalization and localization of the requirement has been removed from the pattern in here. This is further described in the architecture-oriented safety case, which is provided as justification (J5) to the strategy (S6). This justification is “glue” that links the safety requirement argument to the architecture argument. The architecture-oriented safety case identifies how the system safety requirements are broken down into component safety requirements, and where they are ultimately established, thus deriving a hierarchy of requirements that is aligned with the hierarchy of the components in the system. In some sense, the architecture-oriented safety case can be considered as a “subroutine” or away goal to S6: it takes a system-level requirement, and returns a list of formalized component-level requirements, which are then shown in G(FC1). Below this, the argument proceeds as described in the “Formalization of Safety Requirement” pattern (see page 87).

The purpose of the architecture-oriented safety case is to show a compositional verification of the software based on the system architecture and thus provide an independent

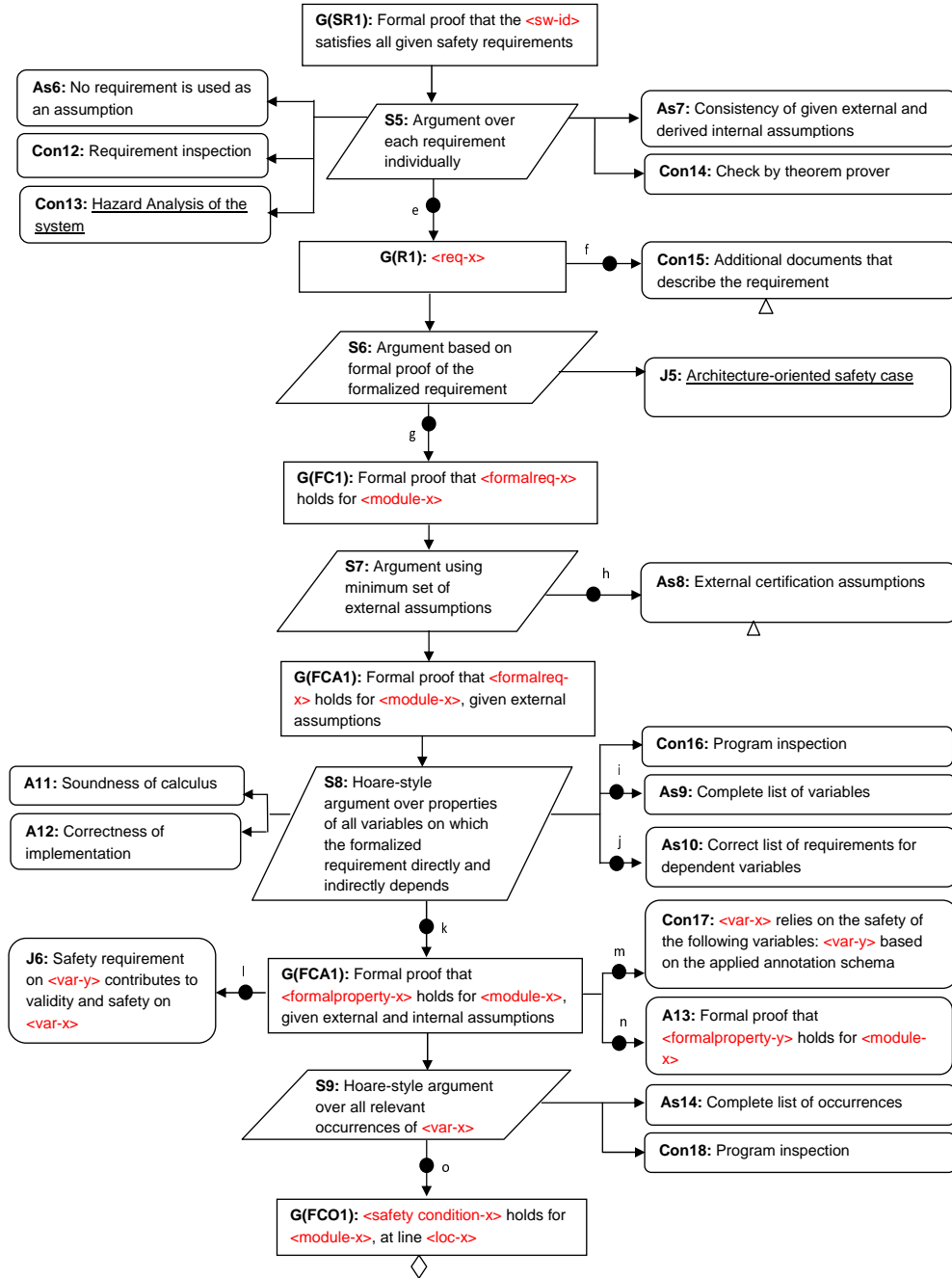


FIGURE 8.3: Tier II of the Heterogenous Safety Case: Arguing over Safety Requirement

assurance of both code and model. The overall argument structure on the architecture-oriented safety case (see Figure 8.4) follows the same argument structure as described in the pattern “Arguing System-Level Safety Requirements to Component-Level Safety Requirements” on page 104. However, the architecture safety case now only serves as abstraction of the actual arguments in the “full” safety case shown in Figure 8.3, providing an additional view on and structuring of the same underlying information. This also subsumes the combination of system-level and component-level safety cases described in

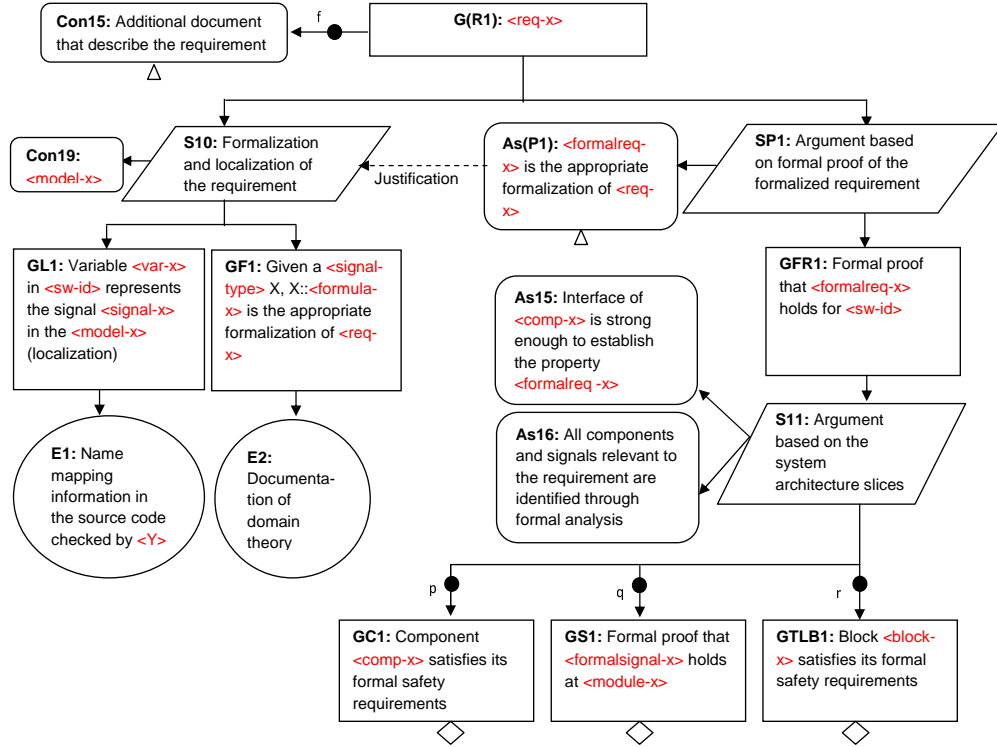


FIGURE 8.4: Tier II of the Heterogenous Safety Case: Arguing over Architecture Slices

Section 6.2.3.

8.2.3 Tier III of Heterogenous Safety Case: Arguing over Sufficiency of Safety Condition

The structure of the argument on tier III of the heterogenous safety case (see Figure 8.5) follows the same lines as shown in the "Sufficiency of Safety Condition" safety case pattern described on page 73 both for safety properties and safety requirements. However, there are some minor modifications to the argument structure in the case that the requirement is established by a library function. This is conceptually similar to the architectural decomposition but since library functions have no corresponding model, we cannot apply our architectural reasoning to gain assurance about their correctness. Instead, we rely on other means, such as testing, to show that the function is correctly specified. We thus add a further argument (G(SC9)) that argues the correctness of the library functions in order to justify that the undesired event E7.1.1 (as identified on page 53) has been controlled and mitigated in the formal program verification. The evidence to show that the library functions are correctly specified against the specification can be attained from the testing results. A reference implementation document that can be used to support the testing processes is described and denoted as a model to the testing strategy. The testing evidence can be double-checked and inspected by subject

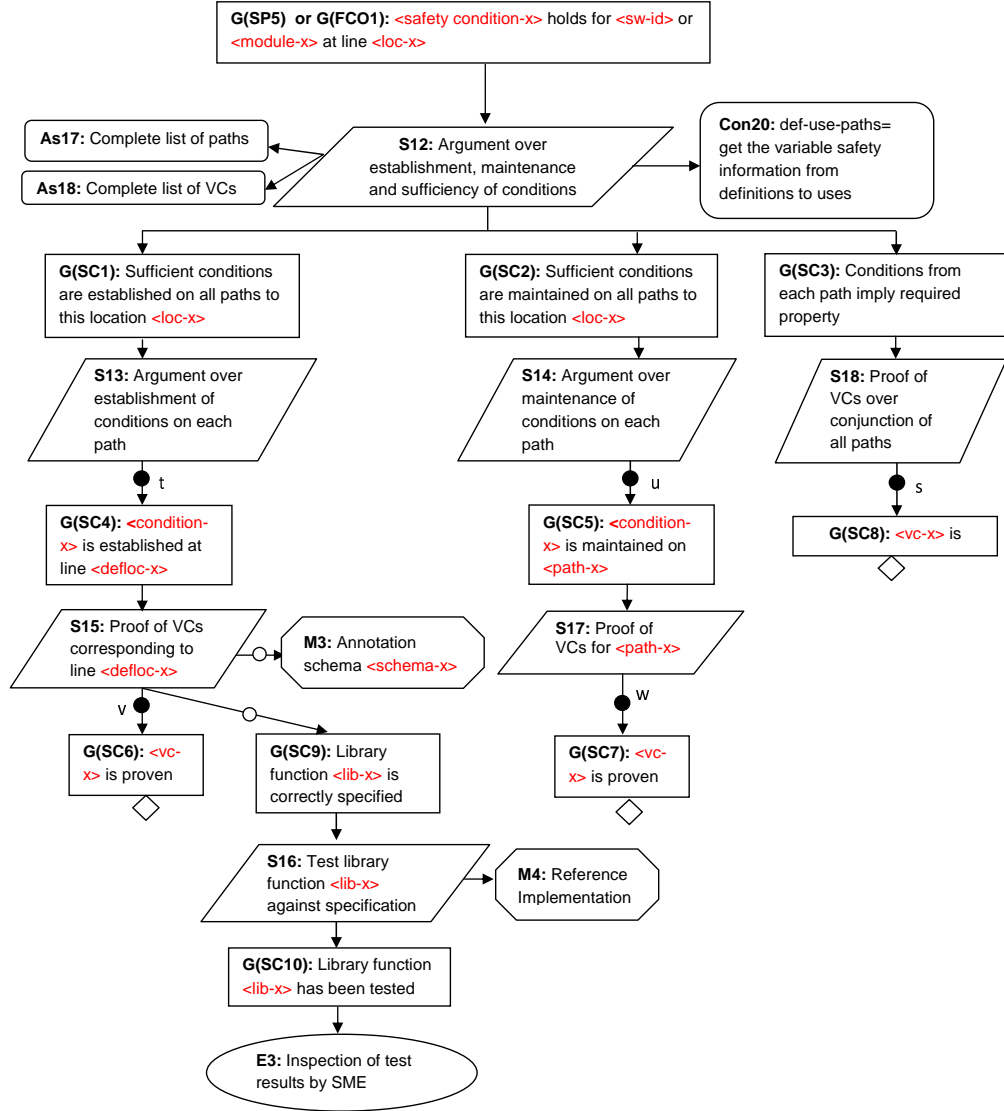


FIGURE 8.5: Tier III of the Heterogenous Safety Case: Arguing over Sufficiency of Safety Condition

matter experts (SME). Note that our formal verification approach now depends on the validity of a non-formal argument, but this can be integrated seamlessly into a single safety case.

8.2.4 Tier IV of Heterogenous Safety Case: Arguing over Soundness of Formal Proof

In formal program verification, automated theorem provers are typically used to automatically prove the VCs. The soundness, correct configuration, and correct installation of the prover should be justified in order to establish trust in the proofs provided. However, the most likely source of invalid proofs are errors in the axiomatization of the domain theory, either logical inconsistencies or inadequate formalizations, and the use

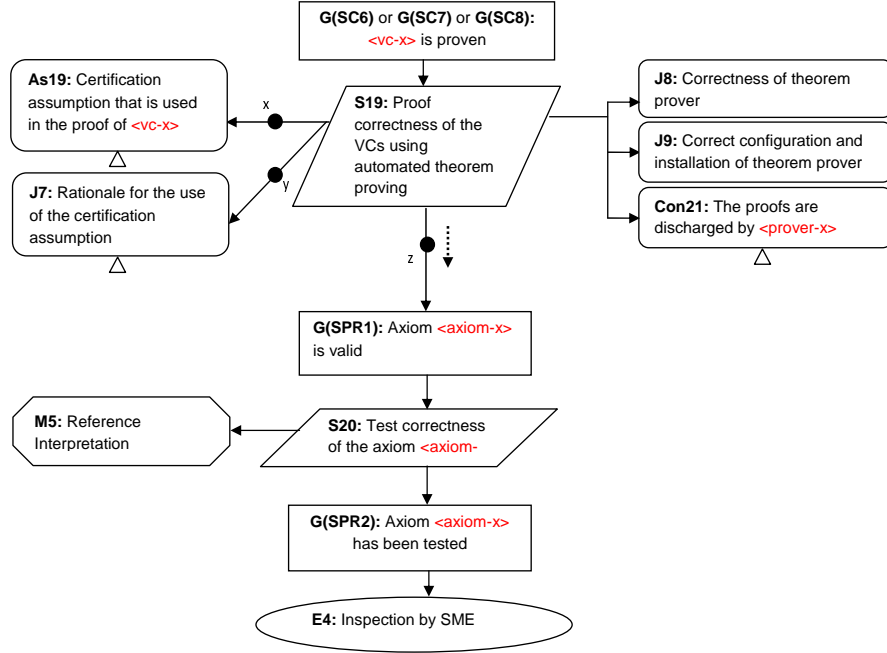


FIGURE 8.6: Tier IV of Derived Safety Case: Arguing over Soundness of Formal Proof

of invalid certification assumptions. In Chapter 7, we have described an approach to construct safety case that correspond to natural deduction (ND) proofs found by the Muscadet prover [118]. However, these safety case quickly grow very large, despite a number of possible abstraction. Here, we instead simplify the overall proof argument by a single strategy (S19) that represents the deductive reasoning of the VCs into further subgoals until it leads to the axioms. We also highlight the use of the external certification assumptions in order to check the validity of their use in deriving the proofs. The validity of the proofs can then be argued by assuring that the certification assumptions hold (cf. As19 and J7 in Figure 8.6), and by showing that the axioms that are used to prove the VCs are valid. The validity of the axioms can be shown by the testing activities such as described in [21]. The reference interpretation document that is used as a guidance in proving the soundness of the axioms can be specified as a model that is linked to the strategy. The validity of the axioms can also be double checked and inspected by SME. The structure of this argument is shown in Figure 8.6; this justifies that the undesired event E7.1.2 (see FTA Figure 3.4) has been controlled and mitigated.

8.3 Safety Case Evidence

In building convincing safety arguments it is important to demonstrate the suitability of the evidence for satisfying a claim. When selecting the required evidence, it is important to identify different types of evidence so that the requirement or claim can be satisfied with confidence; it is also necessary to demonstrate that enough evidence has been

presented. Showing the suitability of evidence can be made by assessing how well the evidence assures the claim. Evidence attributes can be used to measure the degree of confidence provided by the evidence [87, 146, 147]. There are three evidence attributes that have been defined to show the sufficiency of the evidence presented. The attributes are as follows [87, 146, 147]:

- relevance, i.e., the extent to which an item of evidence entails the claim;
- trustworthiness, i.e., the perceived ability to rely on the character, ability, strength or truth of the evidence;
- independence, i.e., the extent to which complementary items within a set of evidence fulfil the claim.

We use the evidence attributes as a guideline to assess the sufficiency and relevancy of the evidence selected. We identify any other additional evidence or documents that are potentially relevant to support and describe the evidence. In our work, formal proofs are provided as the ultimate evidence for the assurance claims. However, there is a non-zero probability that these proofs are not in fact valid [100]. Greater confidence in the assurance claim can be placed if the rationale behind validity of these proofs can be shown. Therefore, in order to raise the degree of confidence that can be placed in the evidence presented in our safety cases, additional forms of supporting evidence are provided. This supporting evidence retrieved from other verification activities, such as testing, and additional documents such as architecture and design document. The evidence has been spliced in to the core argument structure of our safety cases at the appropriate locations. This additional evidence not only increases confidence in the claim provided, but also represents knowledge about the system and the certification approach presented. For example, the believe in the soundness of the formal proofs can be increased if the axioms that are used to prove the theorems are shown to be valid which can be achieved by providing the appropriate testing results. Here, the testing results can be used as supporting evidence to the soundness of the formal proofs. In our work, we categorize the evidence into two types:

- *formal* evidence, e.g., verification conditions, formal proofs, and axioms; and
- *informal* evidence, e.g., testing, documentation and manuals, or expert judgments.

Both types of evidence can be used as direct evidence to the claim or as supporting evidence. For example, DO-178B states that formal methods are complementary to testing, and hence indirectly implies that evidence generated from formal methods cannot be used as the sole means for compliance with verification objectives [61]. However,

different software standards have different preferences and Defence Stan 00-56 [113] considers formal methods to offer the strongest form of evidence. In our work, we follow the lines of the latter. We thus provide testing and inspection results as a means to support the evidence provided by the formal analysis. However, our focus is on the formal argument, and we explicitly do not integrate it with an argument based on testing of the software; this work remains for future research.

The use of formal analysis in showing software safety has been investigated in the literature. Habli *et al.* [69] discuss the complementary role of formal mathematical arguments in achieving confidence in the safety and reliability of software systems. In addition, Habli and Kelly [68] present a generic safety case that can be instantiated to facilitate the presentation and justification of formal analysis. Galloway *et al.* [62] present a meta-approach for generating arguments for substituting testing with more cost-effective processes, in particular proof-based verification.

Our work falls into the same context but we justify the use of formal program analysis by constructing the safety case to establish trust in the formal proofs as evidence on the safety of the code. We make clear the interactions underlying the proofs of the code which gives us greater confidence in the assurance claim. We provide explicit arguments on the use of the Hoare-style program verification in demonstrating software safety and have successfully applied our framework to NASA applications.

8.4 Conclusions

Demonstrating the correctness of large and complex software-intensive systems in certification requires marshalling large amounts of diverse information, including requirement documents, system architecture, models of the system, source code, and verification and validation artifacts such as VCs, formal proofs and testing results. An integration of these diverse artifacts is essential to explicitly represent the rationale underlying the certification process of the program and thus, establish trust and confidence on the assurance provided.

In this chapter, we have described the overall structure of a heterogenous safety case. We show how we can systematically combine diverse types of information from heterogeneous sources into a single integrated safety case. The core argument structure of the safety case is generated from a formal analysis of automatically generated code, based on automated theorem proving, and driven by a set of formal requirements and assumptions. This is then extended by separately specified auxiliary information giving contexts, assumptions, justifications, and constraints, or additional forms of evidence derived from other verification activities, such as testing. So far we have automatically constructed property-oriented, requirement-oriented, architecture-oriented and proof-oriented safety cases. For future work, we plan to automatically construct the heterogenous safety case

described here. We hope our work will promote the use of formal mathematical arguments in achieving confidence in the software safety. We also hope it will increase confidence in the use of formal methods and also in the use of code generators in safety-critical applications. However, as the integration of all safety cases produces larger safety cases and increases the complexity, we plan to use an abstraction mechanisms to highlight different aspects of the integrated safety case. In particular, we plan to derive a safety case for a specific requirement or a specific subsystem. We also plan to use abstractions to construct minimal but consistent safety case slices that start from all nodes representing specific categories of information, e.g., verification tool information, or testing results. So far, we have introduced an abstraction mechanism to handle proof arguments but left an abstraction of the whole safety cases for future work.

Chapter 9

Implementation and Preliminary Assessment of the Approach

Chapter 9 describes the implementation and preliminary assessment of our approach and summarizes our experience in applying it to code generated by an academic and a commercial code generator, respectively. A checklist is described for whether the constructed safety cases provide a convincing and valid argument that the program is adequately safe for a given context in a given environment.

9.1 Introduction

In our thesis proposition (cf. Section 1.4, page 6) we stated our goal as “to develop an approach to automatically construct safety cases from a formal analysis, based on automated theorem proving, of the automatically generated code”. In order to assess to which extent we have achieved our goal, we need to assess

- whether it is feasible to automatically construct the safety cases based on the formal program verification information;
- whether the claim-argument-evidence structure is acceptable; and
- whether the approach is applicable to industrial settings.

We demonstrate the principal feasibility of the approach by an implementation, and we describe this in Section 9.2. To conclude that the claims deserve sufficient confidence and are convincing, the arguments has to be well-structured, the right evidence has to be used, the assumptions have to be acceptable and the reasoning has to be consistent. We demonstrate this by a means of a safety case checklist, which is described in Section 9.3. Further evidence is given by the acceptance of our safety case in the research community.

So far, we have presented our work in several conferences, workshops and during the GSN User Club Meeting and Adelard ASCE Meeting and also during the internship at NASA Ames Research Center. Our approach has also been successfully applied to AutoFilter Program Synthesis Tool [153] and MathWorks Real-Time Workshop [52] code generators based on the information provided by the AutoCert system that has been developed at NASA Ames.

Unfortunately, it is not possible to completely demonstrate the practical application of our approach in wide areas within the timescale of a doctoral programme, so the answer to this question remains somewhat inconclusive. However, we have demonstrated the feasibility of the approach to a certain level by applying it to industrial applications (i.e., to code generated by the AutoFilter system and Real-Time Workshop for NASA projects) and by checking whether all undesired events as identified during the fault tree analysis have been reasoned and argued. We also have successfully applied our safety case patterns to automatically generate the safety cases for different property-oriented, requirement-oriented, architecture-oriented and proof-oriented safety cases.

9.2 Automating the Safety Case Generation

The development and acceptance of a safety case is a key element of safety regulation in most safety-critical sectors [113, 145]. However, to the best of our knowledge, safety cases are largely constructed manually (see for example [56, 99]) as no advance tool is available to support a more automatic safety case construction: most existing safety case construction tools only provide basic drawing support in a “boxes and arrows” style. For example, GSN: ASCE v3.5 from Adelard [9] is a graphical tool for creating the nodes and links of a safety argument based on a “drag-and-drop” interface. Similarly, the University of York’s Freeware Visio Add-on and the GSNCasemaker [10] are Microsoft Visio based software application that are compatible and compliant with the Microsoft standard “drag-and-drop” interface. Cockram and Lockwood in [38] have described several problems that arise in using a traditional, manual construction of safety cases. This is an obviously time-consuming and expensive process, which will dramatically slow down the construction of the safety case when we are dealing with large amounts of artifacts. Obviously, tools supported by automated analyses are needed to deal with a complex and huge safety argument. An automated safety case construction is also needed when we are dealing with iterative software development and code generation, as it can help to reduce time and cost in constructing the safety case.

As to be expected, the safety cases described in Chapter 4-8 quickly become too large for manual development, even if all abstractions are applied. Fortunately, the bulk of the argument is based on information provided by AutoCert system’s formal program analysis, and the argument structure follows the program and analysis structure, so that

a largely automated safety case construction is possible, and we integrate the analysis with an existing commercial safety case tool to do so. However, some information cannot be provided by the program analysis, such as environment constraints, external assumptions, lists of related documents, or model names. This information must be specified externally by a safety engineer. This also applies to the formalization of the top-level safety requirements that drive AutoCert’s analysis, and their integration with the system-wide hazard analysis and safety case.

In order to support the automated safety case construction, we integrated AutoCert’s formal program analysis with Adelard’s ASCE v3.5 tool [9]. AutoCert has been extended to extract the manually specified information from its own input and to structure this together with all information derived by the analysis into an XML format. The XML file records all the relevant information needed for the safety case construction. Subsequently, an XSLT program is used to transform the XML information into a second XML format called GSN-XML that logically represents the structure of the safety case as defined by safety case patterns described in the previous chapters. The patterns were specifically designed so that the same argument structure can easily be adapted to other programs and systems. Finally, we use a custom Java program to present the safety case using GSN. The Java program helps to set the position of the nodes in the safety case which involved some mathematical calculations and to represent the argument to follow the standard Adelard ASCE file format. This architecture avoids a tight integration of the analysis (i.e., AutoCert) and presentation (i.e., ASCE) tools, and provides enough flexibility to change the latter with little effort. Figure 9.1 summarizes the framework of our safety case generation process. Unfortunately, the print quality of these large safety cases is insufficient for presentation, so we choose to recreate them in Microsoft Word in this thesis.

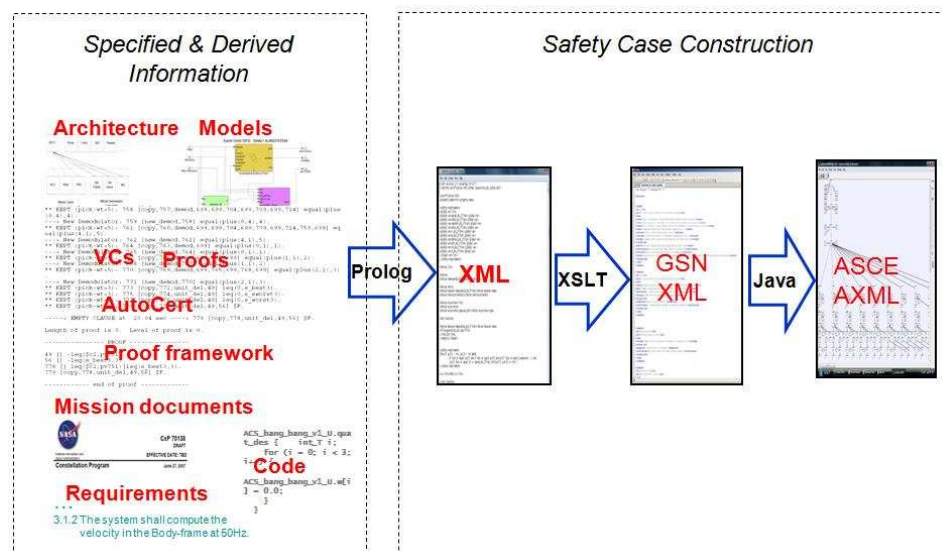


FIGURE 9.1: Safety Case Generation

9.2.1 Architecture of the Safety Case Generation

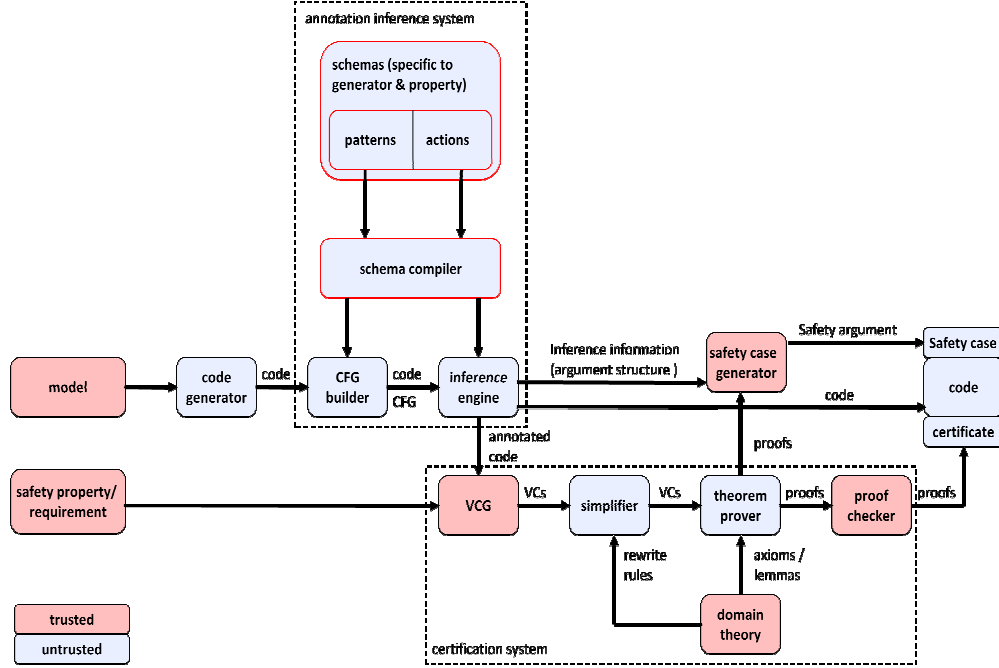


FIGURE 9.2: Extended FSSC System Architecture

The Hoare-approach to safety certification is more flexible than special-purpose static analysis tools such as PolySpace [18] that can only handle the comparatively simple language-specific properties. It also provides explicit evidence in the form of proofs, which static analysis tools typically lack. In our work, we use the Hoare-style program verification, in particular the formal software safety certification system (FSSC) to verify the program safety. FSSC relies on annotations, i.e., logical assertions of program properties, at key locations in the code which are constructed by an annotation inference algorithm. We construct the safety cases based on information from the annotation inference algorithm and let it drive their core argument structure. Therefore, in the existing architecture of the FSSC (cf. Figure 2.4), we integrate the safety case generator with the annotation inference engine in order to receive the annotation inference information. In the extension, the inference engine will supply the information to the safety case generator in form of an XML file. The safety case generator will identify each part of the program that can draw attention to potential certification problems and select appropriate evidence to demonstrate the correctness of the underlying safety claims and the certification process. Figure 9.2 shows the FSSC system [43, 46, 47] extended with the safety case construction. The overall architecture of the safety case generation follows the same line as the FSSC system with the trusted components (in pink) and untrusted components (in light blue). Here, the safety case generator is considered a trusted component: while errors in the safety cases cannot invalidate the proofs (as they

only provide a high-level traceable argument on how the code complies with the specified safety property and requirement). Any errors in the safety case construction can lead to errors in the safety cases, and invalidate the explanation. However, we believe that by elucidating the reasoning behind the certification process using the safety cases, there is less need to trust the original code generation and certification tools.

9.2.2 Implementation

This section describes the XML data format, XSLT program, and Java program that were used to automatically construct the safety cases. Here, we only show some of the program fragments that were used in the implementation. The implementation of the Prolog program that collects the basic information during the automatic annotation inference, and converts it into the XML file, is not shown here, as this is part of the AutoCert system.

```
<?xml version="1.0" encoding="UTF-8"?>
<program-info>
  <hotvar>
    <property-name> frame-safety </property-name>

    <hotvar-name> Quat3 </hotvar-name>
    <hotvar-num-entries> 1 </hotvar-num-entries>

    <hotvar-entry>
      <hotvar-basevar-name> Quat3 </hotvar-basevar-name>
      <hotvar-num-occurrences> 1 </hotvar-num-occurrences>

      <hotvar-occurrence-list>
        <hotvar-occurrence>
          <occurrence-type> def_use </occurrence-type>

          <source-location>
            <file> Nav.cpp </file>
            <line> 65-67 </line>
            <length> 1 </length>
            <schema> compute_dcm_ned_body </schema>
          </source-location>

          <safety-requirement> Quat3:: quat (ECI, Body) </safety-requirement>

          <dvar-list>
            <dvar> Quat2 </dvar>
            <dvar> Azimuth </dvar>
            <dvar> Heading </dvar>
            <dvar> Geolat </dvar>
            <dvar> Geoheight </dvar>
            <dvar> Long </dvar>
          </dvar-list>

        </hotvar-occurrence>
      </hotvar-occurrence-list>
    </hotvar-entry>
  </hotvar>
</program-info>
```

FIGURE 9.3: XML: Program Certification Information

XML Documents. The XML document in Figure 9.3 shows a fragment of the formal program verification information that has been automatically derived during the

construction of the logical annotations. For example, Figure 9.3 describes the safety information of the variable *Quat3*. The formal proof for this variable requires that the formalized safety requirement $Quat3 :: quat(ECI, Body)$ holds for Nav.cpp at a single location, lines 65-67, based on the applied annotation schema *compute_dcm_ned_body*. The validity of safety requirement for Quat3 relies on the safety requirements of variables Quat2, Azimuth, Heading, Geolat, Geoheight and Long. This list of dependent variables are recorded as “dvar” in the XML document. Information about the paths and the generated VCs is described similarly.

```
<?xml version="1.0" encoding="UTF-8"?>

<proof-info>

  <rule>
    <name> ! </name>
    <createobjects>
      <object> z3 </object>
      <object> z2 </object>
      <object> z1 </object>
    </createobjects>
    <newconcl>
      <proof> 0 </proof>
      <id> 2 </id>
      <formula> inc(z1, z2) ^ inc(z2, z3) -> inc(z1, z3) </formula>
    </newconcl>
    <because>
      <because-step>1</because-step>
      <because-reason> concl((0, ![A,B,C]: (inc(A,B) ^ inc(B,C) -> inc(A,C))), 1) </because-reason>
    </because>
    <explanation> the variables of the conclusion are instantiated </explanation>
  </rule>

  <rule>
    <name> -> </name>
    <addhyp>
      <proof> 0 </proof>
      <id> 3 </id>
      <formula> hyp(inc(z1, z2)) </formula>
    </addhyp>
    <addhyp>
      <proof> 0 </proof>
      <id> 3 </id>
      <formula> hyp(inc(z2, z3)) </formula>
    </addhyp>
    <newconcl>
      <proof> 0 </proof>
      <id> 3 </id>
      <formula> inc(z1, z3) </formula>
    </newconcl>
    <because>
      <because-step> 2 </because-step>
      <because-reason> inc(z1, z2) ^ inc(z2, z3) -> inc(z1, z3) </because-reason>
    </because>
    <explanation> to prove H -> C, assume H and prove C </explanation>
  </rule>

</proof-info>
```

FIGURE 9.4: XML: Proof Information

The XML document shown in Figure 9.4 describes the proof information derived from the Muscadet prover. Here, the universal quantifier (i.e., which is represented as ! in Muscadet) and implication elimination rules are used in proving the soundness of the theorem $\forall[A, B, C] : (A \subset B) \wedge (B \subset C) \Rightarrow (A \subset C)$. In the XML document, *object*

refers to the arbitrary object used in \forall rule, *id* refers to the current proof step and *formula* refers to the derived sub-theorem of the current proof step. These XML files are produced by a simple extension of Muscadet’s existing proof output routines. Note that the argumentation represented in these files is much less abstract.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:output method="xml" />
  <xsl:template match="/">

    <certification-info>

      <argument>
        <goal>
          Formal proof that <xsl:apply-templates select=" hotvar/hotvar-entry/hotvar-occurrence-
            list/hotvar-occurrence/safety-requirement"/> holds for <xsl:apply-templates select="
            hotvar/hotvar-entry/hotvar-occurrence-list/hotvar-occurrence/hot-location/source-
            location/file"/>, given external and internal assumptions
        </goal>

        <goal-link>

          <subgoal>
            <xsl:apply-templates select=" hotvar/hotvar-entry/hotvar-occurrence-list/hotvar-
              occurrence/safety-requirement"/> holds for <xsl:apply-templates select=" hotvar/hotvar-
              entry/hotvar-occurrence-list/hotvar-occurrence/hot-location/source-location/file"/>, at
            lines <xsl:apply-templates select=" hotvar/hotvar-entry/hotvar-occurrence-list/hotvar-
              occurrence/hot-location/source-location/line"/>
          </subgoal>

          <strategy>
            Hoare-style argument over all relevant occurrences of <xsl:apply-templates select="
            hotvar/hotvar-entry/ hotvar-basevar-name"/>
          </strategy>

          <assumption>
            <xsl:apply-templates select=" hotvar/hotvar-entry/ hotvar-basevar-name"/> relies on the
            safety requirement of the following variables: <xsl:apply-templates select="
            hotvar/hotvar-entry/hotvar-occurrence-list/hotvar-occurrence/dvar-list/dvar"/> based on
            the applied annotation schema <xsl:apply-templates select=" hotvar/hotvar-
            entry/hotvar-occurrence-list/hotvar-occurrence/hot-location/source-location/schema"/>
          </assumption>

        </goal-link>
      </argument>
    </certification-info>

  </xsl:template>
</xsl:stylesheet>
```

FIGURE 9.5: XSLT: Transforming Program Certification Information into GSN-XML

XSLT Program. The XSLT program in Figure 9.5 extracts the particular data from the formal program analysis and proof information files as shown in Figure 9.3, and transforms them into another XML document representing the logical structure of the safety case. For example, the command *xsl:apply-templates select="hotvar/hotvar-entry/hotvar-basevar-name"* will select the specific child element (i.e., *hotvar-basevar-name*) and write the matching value (i.e., *Quat3*) to the respective location in the specific field (i.e., *goal*) in the GSN-XML file. The XSLT program in some sense implements the generic safety case patterns described in Section 4.3, 5.3, and 6.3. Any changes to

this boiler-plate texts can be done easily by changing the texts in these rules, while structural changes would require new rules.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="xml"/>
<xsl:template match="/">

<certification-info>

<xsl:when test="name = '!'">
<argument>
<goal> <xsl:apply-templates select="because/because-reason" /> </goal>
<strategy>
Argument over an arbitrary element of the domain (<xsl:apply-templates select="name" />)
</strategy>
<strategy-link>
<justification>
<xsl:apply-templates select="createobjects/object" /> are an arbitrary fresh objects
</justification>
<subgoal>
{<xsl:apply-templates select="addhyp/formula" />} <xsl:apply-templates
select="newconcl/formula" />
</subgoal>
</strategy-link>
</argument>
</xsl:when>

<xsl:when test="name = '->'>
<argument>
<goal> <xsl:apply-templates select="because/because-reason" /> </goal>
<strategy>
Suppose premise is true (<xsl:apply-templates select="name" />)
</strategy>
<strategy-link>
<justification>
<xsl:apply-templates select="addhyp/formula" /> can be used as hypothesis to prove
<xsl:apply-templates select="newconcl/formula"/>
</justification>
<subgoal>
{<xsl:apply-templates select="addhyp/formula" />} <xsl:apply-templates
select="newconcl/formula" />
</subgoal>
</strategy-link>
</argument>
</xsl:when>

</certification-info>
</xsl:template>
</xsl:stylesheet>
```

FIGURE 9.6: XSLT: Transforming Proof Information into GSN-XML

In the proof-oriented case (see Figure 9.6), the transformation from XML to the GSN-XML structure highly relies on the natural deduction rules and Muscadet rule templates as described in Chapter 7. Here, each proof step will be transformed into a number of goal, strategy, subgoal, justification, context or assumption nodes as defined in the proof templates, and each XSLT rule implements one of these transformations.

GSN-XML Document. GSN-XML logically represents the structure of safety cases as defined in the safety case patterns in Chapter 4-8. The GSN-XML in Figure 9.7 and Figure 9.8 show the result of the transformation from XML document to GSN-XML document through XSLT program. In our work, the GSN-XML document will be used

```

<?xml version="1.0" encoding="UTF-8"?>

<certification-info>

  <argument>
    <goal>
      Formal proof that Quat3:: quat (ECI, Body) holds for Nav.cpp, given external and internal
      assumptions
    </goal>

    <goal-link>

    <goal-link>
      <subgoal>
        Quat3:: quat (ECI, Body) holds for Nav.cpp, at lines #65-67
      </subgoal>

    <strategy>
      Hoare-style argument over all relevant occurrences of Quat3
    </strategy>

    <assumption>
      Quat3 relies on the safety requirement of the following variables: Quat2 Azimuth
      Heading Geolat Geoheight Long based on the applied annotation schema
      compute_dcm_ned_body
    </assumption>

    </goal-link>
  </argument>
</certification-info>

```

FIGURE 9.7: GSN-XML: Program Certification Information

```

<?xml version="1.0" encoding="UTF-8"?>

<certification-info>

  <argument>
    <goal>  $\neg[A, B, C]: (inc(A, B) \sqcap inc(B, C) \rightarrow inc(A, C))$  </goal>
    <strategy> Show for any arbitrary object (!) </strategy>
    <strategy-link>
    <justifications> z3 z2 z1 are an arbitrary objects </justifications>
    <subgoal> { }  $inc(z1, z2) \wedge inc(z2, z3) \rightarrow inc(z1, z3)$  </subgoal>
    </strategy-link>
  </argument>

  <argument>
    <goal>  $inc(z1, z2) \wedge inc(z2, z3) \rightarrow inc(z1, z3)$  </goal>
    <strategy> Show temporary hypothesis implies goal (->) </strategy>
    <strategy-link>
    <justifications>
       $hyp(inc(z1, z2)) \ hyp(inc(z2, z3))$  can be used as hypotheses to prove  $inc(z1, z3)$ 
    </justifications>
    <subgoal> {  $hyp(inc(z1, z2)) \ hyp(inc(z2, z3))$  }  $inc(z1, z3)$  </subgoal>
    </strategy-link>
  </argument>

</certification-info>

```

FIGURE 9.8: GSN-XML: Proof Information

as an input to automatically generate the safety cases using a Java program.

Java Program. A Java program reads the GSN-XML document and structures the GSN-XML information to follow the standard Adelard ASCE file format. The main

reason to use Java rather than XSLT for this final transformation step is that it requires the computation of layout information that is tedious to implement in XSLT. The position of each node on the ASCE canvas is set based on the calculation of the horizontal and vertical position.

```
// Print ASCE Node
NodeList nodeList = doc.getElementsByTagName("argument");

for (int p = 0; p < nodeList.getLength(); p++) {
    Node fstNode = nodeList.item(p);
    Element fstElmnt = (Element) fstNode;
    NodeList fstNmElmntLst = fstElmnt.getElementsByTagName("goal");
    Element fstNmElmnt = (Element) fstNmElmntLst.item(0);
    NodeList fstNm = fstNmElmnt.getChildNodes();
    String goal = ((Node) fstNm.item(0)).getNodeValue().trim();
    out.println("<node reference=\"G\" + p + \">");
    out.println("<layout x=\"\" + (xpost) + \"\" y=\"\" + (ypost) + \"\" height=\"\" + (node_height) + \"\" width=\"\" + (node_width) + \"\"/>");
    out.println("<ctype>1</ctype>");
    out.println("<user-id><![CDATA[G\" + p + \"]]></user-id>");
    out.println("<user-title><![CDATA[\" + goal + \"]]></user-title>");
    out.println("<status-fields>");
    out.println("<status-field type=\"boolean\" name=\"hasexternalreference\"><![CDATA[False]]></status-field>");
    out.println("<status-field type=\"boolean\" name=\"requiresdevelopment\"><![CDATA[False]]></status-field>");
    out.println("<status-field type=\"boolean\" name=\"requiresinstantiation\"><![CDATA[False]]></status-field>");
    out.println("<status-field type=\"boolean\" name=\"completed\"><![CDATA[False]]></status-field>");
    out.println("<status-field type=\"string\" name=\"resourced\"><![CDATA[]]></status-field>");
    out.println("<status-field type=\"long\" name=\"risk\"><![CDATA[]]></status-field>");
    out.println("<status-field type=\"string\" name=\"confidence\"><![CDATA[Off]]></status-field>");
    out.println("</status-fields>");
    out.println("<html-annotation><![CDATA[");
    out.println("<p>&nbsp;</p>></html-annotation>");
    out.println("</node>");
}
```

FIGURE 9.9: Code: Printing ASCE Node

For example, Figure 9.9 illustrates a program to print the ASCE node. Here, the first statement of the code, i.e., `getElementsByTagName("argument")` returns all child and nested child elements within the specified tag name (i.e., `argument`). The for-loop statement continues to run until the number of nodes (i.e., `goal`) in the argument is complete. Each ASCE node requires a value for the node reference, layout, type of node, user-id, user-title, status-fields and HTML-annotation, as described in ASCE's GSN schema [7]. In ASCE, the different nodes have different node types. For example 1-goal, 2-solution, 3-strategy, 4-assumption, 5-justification, 6-context, 7-model, 8-nodes and 9-option. We introduce constraint as a new node type (i.e., 10-constraint) on top of the existing ASCE node types. Figure 9.10 shows the output generated by this program.

Figure 9.11 shows a program to create a link between ASCE nodes. Each link should contain information about the link reference, type of the link (e.g., 1-issolvedby, 2-incontextof, 3-iteration0to1 and 4-iterationn), strength, source-reference, destination-

```

<node reference="G0">
  <layout x="10245" y="75" height="1500" width="2500"/>
  <type>1</type>
  <user-id><![CDATA[G0]]></user-id>
  <user-title><![CDATA[Formal proof that Quat3:quat(ECI, Body) holds for Nav.cpp, given
external and internal assumptions]]></user-title>
  <status-fields>
    <status-field type="boolean" name="hasexternalreference"><![CDATA[False]]></status-field>
    <status-field type="boolean" name="requiresdevelopment"><![CDATA[False]]></status-field>
    <status-field type="boolean" name="requiresinstantiation"><![CDATA[False]]></status-field>
    <status-field type="boolean" name="completed"><![CDATA[False]]></status-field>
    <status-field type="string" name="resourced"><![CDATA[]]></status-field>
    <status-field type="long" name="risk"><![CDATA[]]></status-field>
    <status-field type="string" name="confidence"><![CDATA[Off]]></status-field>
    <status-field type="boolean" name="filelink"><![CDATA[False]]></status-field>
  </status-fields>
  <html-annotation><![CDATA[ <p>&nbsp;</p> ]]></html-annotation>
</node>

```

FIGURE 9.10: ASCE Node Information

reference and attachment position [7]. Figure 9.12 shows the output generated by this program.

9.3 Evaluation through Safety Case Checklist

We provide a safety case checklist for checking

- whether we have shown that all faults as identified during the fault tree analysis in Chapter 3 have been reasoned and argued in the safety cases;
- whether we have constructed a convincing argument and provided valid evidence for the software safety;
- whether we have not violated any safety case rules.

In the safety case checklist, we identify the locations where the different safety cases (i.e., property-oriented, requirement-oriented, architecture-oriented and proof-oriented) have dealt with these three issues (i.e., the location on where the issues have been shown to be reasoned and controlled). We have shown that we have provided a convincing and valid argument that the program is adequately safe for a given context in a given environment

```

// Print ASCE Link

NodeList nodeList = doc.getElementsByTagName("argument");

for (int z = 0; z < nodeList.getLength(); z++) {
    Node fstNode = nodeList.item(z);
    Element fstElmnt = (Element) fstNode;
    NodeList lstNmElmntLst = fstElmnt.getElementsByTagName("strategy");

    for (int x = 0; x < lstNmElmntLst.getLength(); x++){
        Node lstNmElmnt = lstNmElmntLst.item(x);
        NodeList lstNm = lstNmElmnt.getChildNodes();
        String strategy = "";
        for (int c = 0; c < lstNm.getLength(); c++)
            strategy = ((Node)lstNm.item(c)).getNodeValue().trim();
        if(strategy.equals("") || strategy.equals("\r")) {
            continue;
        }
        out.println("<link reference=\"L\" + \"G\" + z + \"S\" + z + \"\">");
        out.println("<type>1</type>");
        out.println("<strength>1</strength>");
        out.println("<source-reference>S\" + z + \"</source-reference>");
        out.println("<destination-reference>G\" + z + \"</destination-reference>");
        out.println("<attachment x-source=\"0.503\" y-source=\"0\" x-destination=\"0.503\" y-destination=\"1\"/>");
        out.println("</link>");
    }
}

```

FIGURE 9.11: Code: Setting Link of ASCE Nodes

```

<link reference="G0S0 ">
<type>1</type>
<strength>1</strength>
<source-reference>S0</source-reference>
<destination-reference>G0</destination-reference>
<attachment x-source="0.503" y-source="0" x-destination="0.503" y-destination="1"/>
</link>

```

FIGURE 9.12: ASCE Link Information

based on the formal program analysis proofs or other verification and validation evidence. We also check the argument structure of the safety cases to ensure they follow the standard guidelines as described in [86]. Table 9.1 shows the safety case checklist to audit the completeness and consistency of the safety cases wrt. the fault tree analysis and Table 9.2 illustrates the safety case checklist to audit the structure of the safety cases.

TABLE 9.1: Safety Case Checklist: Fault Tree Analysis (adapted from [55])

	Description	Yes	No	Example	Page
Fault Tree Analysis					
Have all undesired events that might invalidate the safety claim construction been cleared?					
F1	Missed potentially unsafe location in the program				
E1	Incomplete or incorrect formalization and localization of requirement	Show evidence on correct formalization and localization of the requirement	✓	S10: Arguing over formalization and localization of the requirement	135
E2	Incomplete program coverage				
E2.1	Missing variables	Assume complete list of variables	✓	As4: Complete list of variables	133
E2.2	Missing variable occurrences	Assume complete list of occurrences	✓	As5: Complete list of occurrences	133
E2.3	Missing paths	Assume complete list of paths	✓	As17: Complete list of paths	136
E3	Missing VCs	Assume complete list of VCs	✓	As18: Complete list of VCs	136
E4	Missing or incorrect critical annotation	Correct annotation is constructed at key program locations	✓	S12: Argument over establishment, maintenance and strength of variable safety)	136
F2	Erroneous conclusion that all locations in the program are safe				
E6	Incorrect hypotheses at the particular location	Show the validity of establishment, maintenance and sufficiency of hypotheses in deriving the proofs	✓	S12: Argument over establishment, maintenance and sufficiency of safety conditions	136
E6.1	Incorrect hypothesis form definition	Show hypothesis is correctly established on the correct path	✓	G(SC1): Sufficient conditions are established on all paths	136
E6.1.1	Definition on incorrect path	Show the correct definition from the correct path is used			
E6.1.2	Definition has incorrect annotation	Show definition is constructed correctly at the correct location			
E6.2	Hypotheses not maintained correctly along paths	Show hypotheses are maintained from def-location to use-location	✓	G(SC2): Sufficient conditions are maintained on all paths	136
E6.3	Hypotheses for different paths inconsistent	Show consistency of the hypotheses in all paths	✓	G(SC3): Conditions from each path imply required property	136
E7	Invalid proof	Show validity of the proof found by automated theorem prover	✓	S19: Prove correctness of the VCs using automated theorem proving	137
E7.1	Invalid domain theory - show correctness of domain theory				
E7.1.2	Invalid axioms	Show correctness of the axioms	✓	S20: Test correctness of the axiom <code><axiom-x></code>	137
E7.1.1	Incorrect library function	Show library function is correctly specified	✓	S16: Test library function <code><lib-x></code> against specification	136
E7.1.3	Invalid certification assumptions	Highlight and justify assumptions that are actually used	✓	S7: Argument using minimum set of external certification assumptions	134
E7.2	Incorrect mechanism - show correctness of certification mechanism				
E7.2.1	Unreliable ATP	Show validity of the proofs found by automated theorem prover	✓	S19: Prove correctness of the VCs using automated theorem proving	137
E7.2.2	Unreliable proof checker	Show validity of the proofs found by automated theorem prover	✓	S19: Prove correctness of the VCs using automated theorem proving	137
E8	Incorrect safety obligation	Show that safety condition holds at particular location	✓	S12: Argument over establishment, maintenance and sufficiency of safety conditions	136
E8.2	Incorrect formalization of safety policy	Prove soundness and completeness of safety policy	✓	G(SCR1): Formalization of safety policies are adequate	131
E8.3	Incorrect safety predicate	Justified that safety condition is derived by instantiation of the safety predicate over occurrences	✓	J4: Safety condition is derived by instantiation of the safety predicate over occurrences	133
E8.1	Incorrect critical annotation	Correct annotation is constructed at key program locations	✓	S12: Argument over establishment, maintenance and sufficiency of safety conditions	136
E8.4	VCG constructed incorrect VCs	Prove correctness of each individual VC by automated theorem proving	✓	S19: Prove correctness of the VCs using automated theorem proving	137

9.4 Industrial Applications

Two real industrial applications have been chosen as examples throughout the course of this research. The first application uses the formal program verification information

TABLE 9.2: Safety Case Checklist: Argument Structure (adapted from [55])

		Description	Yes	No	Example	Page
Argument Structure						
Is the argument structure easy to follow and convincing?						
1.	Is the overall claim clear and unambiguous statement?	Uses Noun-Phrase Verb-Phrase form	✓		GL1: Variable <var-x> in <sw-id> represents the signal <signal-x> in the <model-x> (localization)	135
2.	Is each strategy set out a simple predicate?	Uses standard forms: "Argument over <approach>" "Argument based <approach>" "Argument using <approach>" "Argument of <approach>"	✓		S1: Argument based on proof of partial correctness wrt. the given requirements	131
3.	Is the strategy necessary and sufficient to show the claim is true?	Consider all undesired events that might invalidate the safety claim construction	✓		S5: Argument over each requirement individually	134
4.	Is each branch of argument structure terminated in evidence?	No claim or argument without evidence in the safety case	✓		E1: Name mapping in information in the source code checked by <Y>	135
5.	Is all evidence clear, relevant and conclusive in showing the argument to be true?	The validity of formal proof is supported by other verification and validation activities such as testing (i.e., different forms of evidence to give sufficient confidence)	✓		E3: Inspection of test results by Subject Matter Expert (SME)	136
6.	Is the context clear and sufficient to support the attached node?	Context is used in two forms: "Reference to contextual information" "Statement of contextual information"	✓		Con1: The software consists of the following modules: <modules>	131
7.	Have all assumptions been clearly stated?	Support the claim, strategy and solution.	✓		As6: No requirement is used as an assumption	134
8.	Have all justifications to justify the rationale for the use of particular goal/strategy/evidence been clearly stated?	Describe rationale for the adoption or establishment of goal, strategy and evidence.	✓		J3: Soundness and completeness of safety policy	133
9.	Have all restrictions imposed in which argument can be achieved been clearly stated?	Limitation of the approach or claim.	✓		C2: Certification works on source level representation only	131
10.	Is the level of decomposition of the argument is appropriate? Can it be simplified?	Restructure the resulting safety case to help in emphasizing the essential information.	✓		J5: Architecture-oriented safety case	134
11.	Is the model explicitly described?	Reference to strategy, goal or solution.	✓		M1: Hoare style program verification using specific proof rules	131
12.	Any violation of the safety case approach?					
	Strategies have direct solution	Strategies cannot have direct link to solution. Only subgoals or goals can have direct link to solution.		✓		
	Solutions supported by other elements	Solutions should be the last node of the argument without any further "is-solved-by" link.		✓		
	Missing links (in-context of, is-solved by)	All nodes are linked either with "In-context-of" or "Is-solved-by" links.		✓		

for the code that has been automatically generated by the AutoFilter system developed at NASA Ames Research Center. The second application uses the formal program verification information for code that has been automatically generated for NASA Project Constellation uses Real-Time Workshop for its GN&C systems. In both cases, we use the AutoCert system [52] to support the subsequent certification of the codes created by the code generators. We only concentrate on natural deduction style proofs found for the subsequent certification of both generated codes. The applications of the approach have been described in detail in Chapter 4-8. The list below describes the reasons of why we use the particular applications and tools in the research.

- *Generator* - two different generators in order to show that our approach is independent of the underlying code generator and program. We use an academic (i.e., AutoFilter) and a commercial (i.e., Real-Time Workshop) code generator to assess the flexibility of the approach from a small and simple code to the large and complicated one. Both generators produce code that is substantially different in style.
 - *AutoFilter: Synthesis of State Estimation Software*. AutoFilter is a generator that automatically generates implementations in C or Modula-2 that solves state estimation problems using Kalman filters algorithm.
 - *Mathworks Real-Time Workshop*. Real-Time Workshop is a commercial model-based code generator that generates executable C code from Simulink or Matlab models.
- *Certification Context* - two types of requirements, i.e., safety properties and safety requirements in order to show that our approach is independent of the given certification requirements.
 - *Safety Property*. A safety property is a property stating that “something bad does never happens” during the execution of the program. A safety property is an invariant that needs to hold everywhere in the program. Two general types of safety properties are considered in this work, language-specific properties and domain-specific properties. However so far, we only implement the work to language specific properties. We hope the same derivation can be applied to domain specific properties as well. Language-specific properties concern the safety aspects of the code which depend on the semantics of the programming language, while domain-specific properties concern the use of the code in a particular domain.
 - *Safety Requirement*. A safety requirement is usually related to the functional requirements of the system. It has usually been identified during the hazard analysis of the overall system. Safety requirements are assertions that need to hold at particular occurrence in the program, initially at the end of the program, and thus have more of a liveness “flavor”.
- *Formal Program Verification Approach and Evidence* - we use a formal program analysis, in particular AutoCert system that is based on the Hoare-style program verification approach in order to provide the highest level of assurance for code safety. We believe that formal program verification can provide the highest level of assurance when they are combined with explicit safety arguments as the one we derived here.
 - *AutoCert System*. AutoCert is implemented as a generator-independent plug-in that supports the certification of automatically generated code. It formally

verifies that the auto-generated code is free of certain safety violations and complies wrt. all given safety requirements and safety properties. AutoCert approach is independent of the particular generator used, and need only be customized by the appropriate set of annotation patterns.

- *Natural Deduction Proofs.* We concentrate on natural deduction style proofs, which are closer to human reasoning than resolution proofs, and show how to construct the safety cases by covering the natural deduction proof tree with corresponding safety case fragments. Moreover, the conversion from ND proofs to safety cases is fairly straightforward. In our work, we show how the safety case approach can be applied to the natural deduction proofs found by the Muscadet prover.

9.5 Conclusions

This chapter has described steps that have been used to assess the feasibility of the work presented in Chapter 4-8. It starts with an explanation about the architecture of the safety case generation and shows some pieces of executable code in Java and XSLT that are used to automatically generate the safety cases. We also explain the justifications for why we chose particular industrial applications and tools in the research. Safety case checklists to assess the consistency of the generated safety cases toward the identified hazards in fault tree analysis and to inspect the correctness of the argument structure presented are discussed in this chapter as well. Although it is not possible to evaluate the feasibility of approach more explicitly in wider areas, due to time limitations, we believe the approach of constructing the safety cases from the formal program verification information has been successfully shown to be feasible for arguing the safety of a program. The following chapter summarizes the work presented in this thesis and lists future works related to the research.

Chapter 10

Conclusions

Chapter 10 describes the conclusions, main contributions, and areas of future work.

10.1 Main Contributions

The thesis has described an approach to systematically and automatically derive safety cases from formal program verification information of automatically generated code. The approach has been developed from a conceptual basis to a practical implementation of the concepts using the Goal Structuring Notation. The approach has been successfully illustrated on code generated by the AutoFilter system and the Real-Time Workshop, i.e., ranging from the small and less complicated to large and complicated code. The purpose of the constructed safety cases is to gain confidence in the claims of the formal program verification process by providing a defensible argument for the safety of the software. The contributions of the research presented in this thesis are as follows:

- Analysis of the possible events that might invalidate the assurance claims provided by proofs of the safety of the program.
- Development of the safety cases from information collected during a formal analysis of automatically generated code.
- Introduction of an approach to combine diverse types of information from heterogeneous verification and validation sources into a single, integrated safety case.
- Development of software safety argument patterns which will guide in the construction of the safety cases.
- Introduction of an approach to construct safety cases automatically by integrating the AutoCert formal program verification system with one of the most widely used commercial safety case tools, i.e., Adelard ASCE v3.5.

The following sections elaborate the contributions and draw conclusions related to each of them. They also discuss our contributions in relation to other research.

10.1.1 Fault Tree Analysis

Leveson *et al.* [94, 96, 97] claim that a detailed and dedicated software safety analysis and verification procedure is required to provide extra assurance on the system-safety level. Therefore, they have introduced software fault tree analysis (SFTA) as a method to analyze the safety aspects of software. For example, SFTA has been applied to analyzing the software safety aspects in the product lines [41] and in the assembly language program written in Ada [96]. However, Leveson *et al.* state that SFTA is not a substitute for the verification and validation procedures. Static analysis of the code is thus required for a better assurance [96]. Unlike in other SFTA applications, our work uses a fault tree analysis to identify the hazards that can lead to failures in the program verification system and thus can invalidate its claims about the system safety.

In our work, we need to make the simplified but conservative assumption that every violation of the given requirement is a “potential condition that can cause harm to personnel, system, property or environment” or a hazard because we try to provide some safety assurance already before the full application context of the software is known. A fault tree demonstrates the interaction logic between the undesired events in the program and the certification system, and how this could lead to an undetected violation of the given requirements (i.e., safety requirements and properties). The fault tree analysis focuses on the interaction between the code generator and the certification system. It removes all doubts concerning the correctness (i.e., whether each proof step is legal in the underlying calculus) and validity of the proofs or the correctness of any other certification tools involved in the construction of the assurance claim. Results from the analysis are presented in a tree structure with the undesirable event (i.e., an undetected violation of a safety property or safety requirement) at the root and the causing events at the bottom of the tree.

10.1.2 Approach to Construct Safety Case for the Formal Program Verification of Auto-Generated Code

Nowadays, the development and acceptance of safety argument or safety case is a key element of safety regulation in most of safety-critical sectors [113, 145]. For example, Weaver [146] in his thesis presents arguments that reflect the contribution of software to the safety-critical system and Reinhardt [123] presents arguments over the application of the C++ programming language in the safety-critical systems. Similarly, our approach works in the same context, i.e., presents arguments over the safety of the generated program wrt. the given requirements for the use in the safety-critical systems. Audsley

et al. [26] present arguments over the correctness of the specification mapping, i.e., a translation from the system specifications into a model and subsequently into a code. This is similar to our approach of showing the correct formalization and localization of the requirement.

With the increased use of model-based development in safety-critical applications, the integration of safety cases into such approaches have become an important research topic. For example, Chen *et al.* [35] introduce an integration of model-based engineering with safety analysis and safety cases to help in assessing decisions in system design of automotive embedded systems. Similarly, Wu [157] introduces a framework to facilitate the safe architectural design in safety-critical applications. Hause and Thom [71] describe how SysML and UML can be used to model system requirements and how the safety requirement and other system elements identified in the system design were used to construct the safety case. However, the focus in this research is on extending the modelling framework to represent safety cases using the applied notation. In contrast, in our work, we construct a safety case that argues along the hierarchical structure of systems in model-based design and show how the hierarchy of requirements is aligned with the hierarchical model structure. Rushby [126] also uses automated theorem proving technology (based on the Yices SMT solver) to make a safety argument, but does not construct a detailed safety case. Moreover, his analysis starts with a manually constructed logic-based model of the system, where the connection to the underlying code remains unclear while our approach works directly on the code.

In order to demonstrate a compelling argument on software safety assurance, Hawkins and Kelly in [72] provide a framework for justifying the arguments and evidence required to demonstrate sufficient assurance in the software. Littlewood and Wright [100] state that the probability of a claim, which has been shown by a formal proof, being false, is very low, when the assumptions and evidence are valid. As pointed out by Littlewood and Wright [100], we believe substantial confidence can be placed if the validity of the underlying proof construction can be shown. We also believe a proof-based verification is sufficient to provide assurance on safety of the program. Other work supports our view. Galloway *et al.* [62] present arguments for technology substitution, i.e., argue over substitution of testing with a proof-based verification technique in the context of the certification standards such as DO-178B [61], and Habli and Kelly [66, 67, 68] carried out research in constructing a safety argument to facilitate the justification and presentation of formal analysis in supporting testing techniques as presented in the software standards. Similarly, our work falls into the same context, i.e., to justify the application of a formal program analysis method in providing assurance for the software safety and the use formal proofs as an evidence.

However, all of these works [62, 66, 67, 68] remain completely generic and do not take the actual code into account. Our work in contrast focuses on constructing a defensible argument on how specific code complies with the specified requirements (i.e., safety

requirements and safety properties) and how the code can be concluded to be sufficiently safe and correct based on the evidence (i.e., formal proofs and others verification and validation artifacts) available. The core argument structure of the safety case is derived from a formal analysis of automatically generated code, based on automated theorem proving, and driven by a set of requirements and assumptions. We also construct safety cases that recover the system structure and component hierarchy from the code, providing independent assurance of both code and model. This is then extended by separately specified auxiliary information giving contexts, assumptions, justifications, and constraints, or additional forms of evidence derived from other verification activities, such as testing. In addition, we also construct safety cases that correspond to formal proofs found by automated theorem provers and reveal their underlying argumentation structure and top-level assumptions.

10.1.3 Software Safety Argument Patterns

The software safety argument patterns presented in this thesis are a generalization of the detailed arguments for arguing the safety and correctness of software based on a formal analysis of the program. The patterns provide reusable goal structures that can be used for the construction of the property-oriented, requirement-oriented, architecture-oriented and proof-oriented safety cases for concrete programs. Each pattern tackles only one aspect of the overall structure of the safety argument contained within a heterogeneous safety case. Weaver in [146] provides patterns for arguing selection of evidence in assuring safety of the software wrt. the system safety requirements. In his thesis, Weaver distinguishes system safety requirements into safety feature requirements and hazard-based requirements. Safety feature requirements relate to either software safety function and safety property while hazard-based requirements are defined as a system behavior at the system-environment boundary that can lead to unsafe actions (e.g., hardware, software or human failure). The main focus of the arguments in his thesis [146] is on arguing over safety of software wrt. hazard-based requirements. Our work is based on the same objective, i.e., to provide patterns for arguing acceptability of the software safety, but we focus on safety of the program wrt. safety feature requirements, in particular safety property and safety requirement. Moreover, our patterns focus on showing how we can establish trust, from the evidence available (i.e., formal proofs), that the software is safe and correct wrt. all given safety requirements. Our safety case patterns are generic as they can be used as templates in the automatic safety case construction. Some of nodes in the safety case patterns are fixed (i.e., can be reused in new contexts or applications without being changed much from the original). In principle, our safety case patterns are independent of the given requirements and program, and consequently also independent of the underlying code generator, as the general safety considerations in each pattern remain the same.

10.1.4 Automatic Safety Case Generation Approach

Manual safety case construction is impractical, especially when demonstrating the safety and correctness of large and complex software-intensive systems as it requires marshalling large amounts of diverse information, e.g., models, code, specifications, mathematical equations and formulas. The resulting safety cases might consist of various arguments and auxiliary types of evidence and thus, it is hard to trace the relation between the claim, argument and the evidence presented. Cockram and Lockwood in [38] have described several problems in using traditional (manual) safety case construction. Issues such as coordinating and maintaining the complex safety case especially when dealing with iterative software development might weaken the motivation to use the safety case technique. Therefore, tools supported by automated analyses are needed to help in the development of a complex safety case.

Several efforts have been made to improve the safety case construction and to manage the artifacts used in the safety case. For example, eSafety Cases [38] build by Praxis HIS is a tool to automate the creation of a link from the safety case to the related material (in Adobe Acrobat Reader and HTML browsers) by a means of hypertext. This tool uses an intranet technology to provide a hyperlink to the safety case's relevant documents. Fararooy [13] introduces the ISCaDe software package as a means to help in the development and maintenance of a safety case. This software package, however, requires the goals, strategies, justifications and other argument elements to be defined in the safety case module before it can be automatically transformed into the safety case diagram. Our approach, in contrast, not only provides hypertext links to the related resources but also automatically generates the safety case without requiring any of the argument's elements to be defined in advance.

In our work, we introduce an approach to automatically construct the safety case from the formal program verification information. We integrate the AutoCert [52] formal program verification system with an existing safety case construction tool i.e., Adelard's ASCE v3.5 tool [9]. The integration allows us to automatically derive safety cases from the information collected during the formal program verification phase together with some specified verification and validation information, such as testing results.

10.2 Limitations

While the presented work has successfully been applied to the formal program verification of automatically generated code, it has several limitations that must be acknowledged as below.

1. We assume that the calculus is sound and the formalization of the safety policy is adequate, as we have not yet expanded the corresponding arguments in this work.

Similarly, we assume that the implementation of the VCG, ATP and all auxiliary components used for constructing and proving VCs are correct.

2. We construct the argument based on the safety requirements and external assumptions as given only. Hence, missing safety requirements or invalid assumptions can weaken or invalidate the assurance claims.
3. We rely on the AutoCert system to provide the information underlying the safety case construction. Any limitations (e.g., in the expressiveness) or failures of the AutoCert also limit the applicability and practicability of our approach.
4. We rely on off-the-shelf ATPs to prove the constructed VCs. The full safety case can only be constructed if these do not fail. If they fail to prove some of the VCs, these VCs become assumptions to the safety case. We do not handle the case where the ATPs refute VCs as invalid, which would indicate a violation of the stated safety property respectively safety requirement.
5. We have not yet performed an intensive evaluation of the approach, e.g., using questionnaires.

10.3 Future Work Directions

The future work directions we intend to pursue are outlined below.

1. A complementary safety case that argues the safety of the certification framework, in particular the safety of the underlying safety logic (i.e., the soundness of Hoare-calculus, the language semantic and safety policy) should be constructed.
2. An additional structuring mechanism to manage and present the structure of the generated safety cases, for example, a plug-in to exploit the abstraction feature in the ASCE v3.5 tool, should be developed. As the safety cases are generated automatically, their sizes are highly dependent on the given formal program verification information and the safety cases can easily get too large and complex, i.e., the scalability issue. In order to reduce the complexity and also for a better understanding, a simplify structure should be introduced (such as an away goal to support the backing evidence of the argument) and a plug-in to support the abstraction mechanism should also be provided. The abstraction mechanism should be able to derive safety cases that highlight different safety aspects and construct minimal but consistent safety case of a particular requirement or a particular subsystem.
3. A prototype of the automatic safety case generation for the heterogeneous safety case should be implemented.

4. More tools such as other commercial code generators and formal program verification systems should be integrated with the approach.
5. The approach should be evaluated in more detail, e.g., by integration with any safety case evaluation systems [39] or via questionnaires that assess the clarity and completeness of the safety cases presented and by using a bottom-up analysis to check the completeness of the fault tree analysis results.

10.4 Concluding Remarks

Based on formal proofs, formal methods can in principle provide the highest level of assurance of the code's safety. However, the formal proofs by themselves are no panacea in software safety assurance, and it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claim and the proof rest. We believe that purely technical solutions such as proof checking [156] fall short of the assurance provided by our safety case, since they do not take into account the reasoning that goes into the construction of the VCs. We also believe that a purely manual construction of the safety cases is too slow and too expensive to cope with the challenges posed by iterative and model-based software development approaches. We believe that our approach to automatically instantiate safety case patterns strikes the right balance.

We consider the safety cases as a first step towards a fully-fledged software certificate management system [45]. We believe that the result of this research, that is, a combined safety case (i.e., for the program being certified, an integration with other verification and validation information, as well as the safety logic and the certification system), will clearly communicate the safety claims, key safety requirements, and evidence required to trust the program. We also believe that our approach helps to begin establish trust in the application of proof-based verification in safety-critical systems. We hope that the interest expressed in some of the concepts within this research will lead to industrial application of the ideas, and improvements in the development and assessment of safety-critical software.

Appendix A

Roles and Principles of FSSC Components

TABLE A.1: Roles and Principles of FSSC Components [44, 47, 153]

Category	Component	Roles	Principles
Code Generation (AutoFilter)	Code Generator	Automatically generate and translate code into a targeted platform	<ul style="list-style-type: none"> • Takes high level specification of state estimation tasks and derives a generated code based on variants of the Kalman filter algorithm • Derives code by repeating application of schemas • Based on schema based synthesis approach, i.e., combination of deductive reasoning and generative programming techniques
	Symbolic Computation	Support schema instantiation and code optimization	<ul style="list-style-type: none"> • Small rewrite engine which supports associative-commutative operators and explicit contexts • Implement expression simplification and symbolic differentiation on top of the rewrite engine
	Schema	Represent fundamental building blocks (algorithm) and solution method (transformation) of program	<ul style="list-style-type: none"> • High level macro that can be applied to problem of certain structure for example linear process model • Generic algorithms which can be instantiated in a problem-specific way after the applicability conditions have been proven to hold for the given problem specification • Contains a parameterized code fragment (i.e., template) together with a set of constraints that determine whether the schema is applicable and how the parameters can be instantiated • Similar to the lemmas used in purely deductive systems but schema can contain explicit calls to a meta-programming kernel in order to construct code • Organized hierarchically into schema library which further constrains the search space
	Code Fragment	Represent code in schemas	<ul style="list-style-type: none"> • Can contains parameters that are instantiated by schema application • Code fragments are assembled, optimized and translated into a target platform • Formulated in an intermediate language (see intermediate code)
	Intermediate Code	Represent low level code	<ul style="list-style-type: none"> • Can be translated to different targeted platforms (languages) • Intermediate code is formulated when code fragments are translated into an essential intermediate language • Contains sanitized variant of programming language (e.g., no pointers, no side-effects in expression) and domain-specific (e.g., vector/matrix operations, finite sums, and convergence loops)
	Optimization	Optimize intermediate code	Employ code motion, common sub-expression, elimination and memoization for intermediate code optimization
	Target Code Generator	Represent translation of intermediate code which is tailored for a specific run-time environment	Employ rewrite system to eliminate the constructs of the intermediate language which are not supported by target environment (desugaring) and to clean up the desugared code.

TABLE A.2: Roles and Principles of FSSC Components [44, 47, 153]

Category	Component	Roles	Principles
Annotation Generation	Annotation Schema	Construct and insert the annotations based on the annotation template	<ul style="list-style-type: none"> The annotation schemas are applied using a combination of planning and aspect oriented techniques to produce an annotated program Use techniques similar to aspect oriented programming to add annotations Use patterns to capture the idiomatic code structures and pattern matching to find the corresponding code locations. Annotate the intermediate code with mark-up information relevant to the policy
	Annotation Inference Algorithm	Constructs the annotations by analyzing the program structure	<ul style="list-style-type: none"> To “get the information from definitions to uses”, i.e., to find all program locations where the safety-relevant information will be established or “defined”, as well as all potentially unsafe locations, where it is used, and then to construct the formulae required for the annotations, and to annotate the program along the control flow-paths between definitions and uses. The annotation inference algorithm is generic, and parameterized wrt. a library of coding patterns that depend on the safety policy and the code generator
	Annotation	Formalize auxiliary safety information for the program	<ul style="list-style-type: none"> Extra information associated with a particular point in a code that are required for formal safety proofs Contain local information of the generated code in the form of logical pre and post-conditions, and loop invariants which are the main elements in the certification process
Safety Component	Safety Property	Represent an operational characterization of intuitively safe programs	<ul style="list-style-type: none"> States that something bad does never happen during the execution
	Safety Requirement	Related to the functional requirements of the system	<ul style="list-style-type: none"> Usually been identified during the hazard analysis of the overall system Any possible failure in the system wrt. safety requirements should be controlled and mitigated in order to prevent a system failure Need to hold at specific location in the program
	Safety Policy	<ul style="list-style-type: none"> Hoare-style proof rules and auxiliary definitions Designed to show that safe programs satisfy a safety property of interest 	<ul style="list-style-type: none"> Different properties are formalized by different safety policies in Hoare-logic but can be mixed and matched for certification goal purposes Formalized using the usual Hoare triples (i.e., $\{P\} C \{Q\}$), which are extended with shadow variables and a safety predicate. Consists of two levels of safety policies, language specific properties (e.g., array bounds safety and initialization before use safety) and domain specific properties (e.g., vector-norm safety, matrix symmetry) A safety policy is essential in maintaining a logical safety environment and producing the appropriate safety obligations of the program.
	Safety Predicate	Corresponds to semantic safety conditions and denotes language-specific safety requirements	<ul style="list-style-type: none"> Defined on the variables of the generated program Stated as generic logical formula

TABLE A.3: Roles and Principles of FSSC Components [44, 47, 153]

Category	Component	Roles	Principles
Generate and simplify the VCs	VCG	Generate VCs	<ul style="list-style-type: none"> • Applies Hoare rules of safety policies to generate VCs • Designed to be correct-by-inspection i.e. to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic • Does not implement any simplification • Works backwards through the code and verification conditions are generated at each line that can potentially violate the safety policy
	VC	Reduce program verification problem to logical proof problem	<ul style="list-style-type: none"> • A set of logical formulae generated from the annotated program • Consists of the post and precondition, in terms of generic safety predicates • Can be verified by ATPs • VCs are a detailed low-level representation of the underlying program safety information and the process used to derive it • VCs are typically of the form of: $hyp1 \wedge hyp2 \wedge \dots \wedge hypn \rightarrow conc$ <i>hyp</i> can be loop invariant, index bound, propagated annotation and <i>conc</i> derived from annotated assertion, generated safety condition.
	Simplifier	Simplify VCs before they can be transmitted to ATP	<ul style="list-style-type: none"> • Use rewrite-based simplifier • Reuses a set of rewrite rules specified in the domain theory and the synthesizer for simplification process
	TPPTP2X-Converter	Convert VCs into particular first-order logic syntax of different ATPs and transmit them to ATPs	<ul style="list-style-type: none"> • Directly connected to ATP • Use the TPPTP syntax that allow VCs to be proved by a wide range of the off-the-shelf first-order provers
Produce Proofs	ATP	Find proofs from inference rules of its (ATP) calculus, additional domain theory and set of core axioms together with a collection of dynamically generated axioms.	<ul style="list-style-type: none"> • Implement search procedure for finding proofs of VCs • Use first-order logic ATPs only • Use TPPTP control scripts for the integration of different ATPs which run on local server
	Proof	A sequence of statements that follows certain rules of reasoning	<ul style="list-style-type: none"> • Proofs are serve as certificate which indicate safe or unsafe of the program
	Domain Theory	Enhance ATP capabilities in finding and providing proofs	<ul style="list-style-type: none"> • Designed to support ATP and simplifier capabilities in providing proofs • Consists of fixed axioms, lemmas and rewrite rules and dynamically generated axioms for each proof task
	Proof Checker	Double-check and formally verify proofs produced by ATP	<ul style="list-style-type: none"> • Use a small, verifiable algorithm to ensure validity of proofs

Bibliography

- [1] Code Smith Tool. <http://www.codesmithtools.com>, First Accessed: 13th June 2008.
- [2] DBOW - The Database Object Generator <http://dbow.sourceforge.net/>, First Accessed: 13th June 2008.
- [3] dbQwikSite <http://www.dbqwiksite.com>, First Accessed: 13th June 2008.
- [4] Real-Time Workshop Embedded Coder 5.1. <http://www.mathworks.com/products/rtwembedded/>, First Accessed: 13th June 2008.
- [5] Web Application Code Generator. <http://www.gigaframe.com/software/webapplicationgeneratortnet/tabid/83/default.aspx>, First Accessed: 13th June 2008.
- [6] Wrapper Code Generator for MS SQL Server Databases. http://www.codeproject.com/kb/database/csharp_wrapper.aspx, First Accessed: 13th June 2008.
- [7] Adelard ASCE-GSN 1.2 Schema, First Accessed: 15th April 2008.
- [8] NI MATRIXx 8.1. <http://www.ni.com/matrixx/>, First Accessed: 18th August 2009.
- [9] ASCE v3.5 from Adelard. <http://www.adelard.co.uk>, First Accessed: 1st October 2007.
- [10] CET GSNCasemaker. <http://www.esafetycase.com>, First Accessed: 1st October 2007.
- [11] eSafetyCase from Praxis High Integrity Systems. <http://www.esafetycase.com>, First Accessed: 1st October 2007.
- [12] Failure Analysis Methods, Tools and Services. <http://www.albservice.com/en/fracas/failure-analysis-methods-and-tools.html>, First Accessed: 1st October 2007.
- [13] ISCaDE (Integrated Safety Case Development Environment) from RCM2. <http://www.iscade.co.uk>, First Accessed: 1st October 2007.

-
- [14] SpecTRM: Specification Tools and Requirements Methodology. <http://www.safeware-eng.com>, First Accessed: 1st October 2007.
 - [15] Template-Based Code Generation with Apache Velocity. <http://onjava.com/pub/a/onjava/2004/05/05/cg-vel1.html?page=2>, First Accessed: 1st October 2007.
 - [16] Unified Modeling Language <http://www.uml.org/>, First Accessed: 1st October 2007.
 - [17] SCADE suite KCG- DO-178B Code Generator <http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation>, First Accessed: 1st October 2009.
 - [18] PolySpace Technologies. <http://www.polyspace.com>, First Accessed: 5th February 2008.
 - [19] J.R. Abrial. *The B-Method - Assigning Programs to Meanings*. Cambridge University Press, 1996.
 - [20] National Aeronautics and Space Administration (NASA). Mars Climate Orbiter Crash. *Mars Climate Orbiter Mishap Investigation Board Phase I Report*, November 1999.
 - [21] K.Y. Ahn and E. Denney. Testing First-Order Logic Axioms in Program Verification. To appear in *Proceedings of the 4th International Conference on Tests and Proofs (TAP'10)*, LNCS, Malaga, Spain, July 2010. Springer.
 - [22] J. Aldrich. Lecture Notes: Hoare Logic. *School of Computer Science, Carnegie Mellon University*, 2007.
 - [23] B. Alpern and F.B. Schneider. Defining Liveness. Technical report, Cornell University, Ithaca, NY, USA, 1984.
 - [24] A.R. Anderson and N. Belnap. *Entailment: The Logic of Relevance and Necessity*. Princeton University Press, 1975.
 - [25] P.B. Andrews. Transforming Matings into Natural Deduction Proofs. In *Proceedings of the 5th Conference on Automated Deduction*, LNCS 87, pages 281–292. Springer, 1980.
 - [26] N. Audsley, I. Bate, and S. Crook-Dawkins. Automatic Code Generation for Airborne Systems. In *Proceedings of the IEEE Aerospace Conference*, volume 4, pages 8–15. IEEE, 2003.
 - [27] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, Boston, MA, USA, 2003.

-
- [28] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer, New York, USA, 1994.
 - [29] P. Bishop and R. Bloomfield. A Methodology for Safety Case Development. In *Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-Critical Systems Symposium*, pages 194–203, Birmingham, 1998. Springer.
 - [30] P. Bishop, R. Bloomfield, L. Emmet, C. Jones, and P. Froome. *Adelard Safety Case Development Manual*. Adelard, London, 1998.
 - [31] R.G. Brown and P.Y.C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley and Sons, 3rd edition, 1997.
 - [32] J-L. Camus and B. Dion. *Efficient Development of Airborne Software with SCADE Suite - DO-178B White Paper*. Esterel Technologies, 2003.
 - [33] A. Cawsey. *Explanation and Interaction*. MIT Press, 1992.
 - [34] S. Chaki, A. Gurfinkel, K. Wallnau, and C. Weinstock. Assurance Cases for Proofs as Evidence. In *Proceedings of the Workshop on Proof-Carrying Code and Software Certification (PCC'09)*, pages 23–28, 2009.
 - [35] D.-J. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg F. Törner, and M. Törngren. Modelling Support for Design of Safety-critical Automotive Embedded Systems. In *Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP'08)*, LNCS 5219, pages 72–85. Springer, 2008.
 - [36] D. Chester. The Translation of Formal Proofs into English. *Artificial Intelligence*, 7(3):261–278, 1976.
 - [37] E.M. Clarke and J.M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Survey*, 28(4):626–643, 1996.
 - [38] T. Cockram and B. Lockwood. Electronic Safety Cases: Challenges and Opportunities. In *Proceedings of Safety Critical Systems Symposium*. Springer, 2003.
 - [39] L. Cyra and J. Grski. Expert Assessment of Arguments: A Method and Its Experimental Evaluation. In *Proceedings of the 27th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'08)*, LNCS 5219, pages 291–304. Springer, 2008.
 - [40] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
 - [41] J. Dehlinger and R.R. Lutz. Software Fault Tree Analysis for Product Lines. In *Proceedings of the 8th IEEE International Symposium on High-Assurance Systems Engineering (HASE'04)*, pages 12–21. IEEE Computer Society, 2004.

-
- [42] E. Denney and B. Fischer. Explaining Verification Conditions. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST'08)*, LNCS 5140, pages 145–159.
 - [43] E. Denney and B. Fischer. Correctness of Source-Level Safety Policies. In *Proceedings of the Formal Methods (FM'03)*, LNCS 2805, pages 894–913. Springer, 2003.
 - [44] E. Denney and B. Fischer. Certifiable Program Generation. In *Proceedings of the Generative Programming and Component Engineering (GPCE'05)*, LNCS 3676, pages 17–28. Springer, 2005.
 - [45] E. Denney and B. Fischer. Software Certification and Software Certificate Management Systems (position paper). pages 1–5, Long Beach, California, USA, 2005.
 - [46] E. Denney and B. Fischer. Annotation Inference for Safety Certification of Automatically Generated Code (extended abstract). In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 265–268. IEEE Computer Society, 2006.
 - [47] E. Denney and B. Fischer. A Generic Annotation Inference Algorithm for the Safety Certification of Automatically Generated Code. In *Proceedings of the Generative Programming and Component Engineering (GPCE'06)*, pages 121–130. ACM Press, 2006.
 - [48] E. Denney and B. Fischer. A Verification-driven Approach to Traceability and Documentation for Auto-generated Mathematical Software. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09)*, pages 560–564. IEEE/ACM, 2009.
 - [49] E. Denney, B. Fischer, and J. Schumann. Adding Assurance to Automatically Generated Code. In *Proceedings of the 8th International Symposium on High-Assurance Systems Engineering (HASE'04)*, pages 297–299. IEEE Press, 2004.
 - [50] E. Denney, B. Fischer, J. Schumann, and J. Richardson. Automatic Certification of Kalman Filters for Reliable Code Generation. In *Proceedings of the IEEE Aerospace Conference*, Big Sky, Montana, March 2005. IEEE.
 - [51] E. Denney, J. Power, and K. Turlas. Hiproofs: A Hierarchical Notion of Proof Tree. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, ENTCS 155, pages 341–359, 2006.
 - [52] E. Denney and S. Trac. A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code. In *IEEE Aerospace Conference*, Big Sky, MT, USA, 2008.

-
- [53] G. Despotou. *Managing the Evolution of Dependability Cases for Systems of Systems*. PhD thesis, University of York, 2007.
 - [54] R.V. Engelen, L. Wolters, and G. Cats. CTADEL: A Generator of Multi-Platform High Performance Codes for PDE-Based Scientific Applications. In *Proceedings of the 10th International Conference on Supercomputing*, pages 86–93, Philadelphia, USA, May 1996. ACM Press.
 - [55] EUROCONTROL. Safety Case Development Manual. Technical report, October 2006.
 - [56] EUROCONTROL. Preliminary Safety Case for Enhanced Air Traffic Services in Non-Radar Areas using ADS-B Surveillance. Technical report, December 2008.
 - [57] N. Fenton. The Role of Measurement in Software Safety Assessment. In *Safety and Reliability of Software Based Systems - Twelfth Annual CSR Workshop*, pages 217–248, Bruges, Belgium, 1997. Springer.
 - [58] A. Fiedler. P.rex: An Interactive Proof Explainer. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR'01)*, LNAI 2083, pages 416–420, Siena, Italy, 2001.
 - [59] B. Fischer, T. Pressburger, G. Rosu, and J. Schumann. The AutoBayes Program Synthesis System - System Description. In *Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, pages 168–172, Siena, June 1990.
 - [60] M. Fitting. *First-order Logic and Automated Theorem Proving*. Springer, Secaucus, NJ, USA, 2nd edition, 1996.
 - [61] EUROCAE (European Organisation for Civil Aviation Equipment). ED-12B/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1994.
 - [62] A. Galloway, R.F. Paige, N.J. Tudor, R.A. Weaver, I. Toyn, and J.A. McDermid. Proof vs Testing in the Context of Safety Standards. In *Proceedings of the 24th Digital Avionics Systems Conference (DASC'05)*, volume 2, page 14, 2005.
 - [63] M. Garnacho and M. Prin. Convincing Proofs for Program Certification. In *Proceedings of the International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'08)*, ENTCS 238, pages 41–56, 2008.
 - [64] G. Gentzen. Investigations into Logical Deduction. *American Philosophical Quarterly*, 1(4):288–306, 1964.
 - [65] W.S. Greenwell, E.A. Strunk, and J.C. Knight. Failure Analysis and the Safety Case Lifecycle. In *Proceedings of the 7th Working Conference on Human Error, Safety, and Systems Development*, Toulouse, France, August 2004.

-
- [66] I. Habli and T.P. Kelly. Process and Product Certification Arguments: Getting the Balance Right. *ACM SIGBED Review*, 3(4):1–8, 2006.
 - [67] I. Habli and T.P. Kelly. Achieving Integrated Process and Product Safety Arguments. In *Proceedings of the 15th Safety Critical Systems Symposium (SSS'07)*, pages 55–68, Bristol, UK, 2007. Springer.
 - [68] I. Habli and T.P. Kelly. A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. *Electronic Notes Theoretical Computing Science*, 238(4):27–39, 2009.
 - [69] I. Habli, Z. Stephenson, T.P. Kelly, and J.A. McDermid. Software Assurance Arguments vs. Formal Mathematical Arguments: A Complementary Role. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering (ISSRE'09)*, Mysuru, India, November 2009.
 - [70] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.
 - [71] M.C. Hause and F. Thom. Integrated Safety Strategy to Model Driven Development with SysML. In *Proceedings of the 2nd IET International Conference on System Safety 2007*, CP532, pages 124–129, 2007.
 - [72] R.D. Hawkins and T.P. Kelly. Software Safety Assurance - What is Sufficient? In *Proceedings of the 4th IET International Conference on System Safety*, London, UK, 2009.
 - [73] C.L. Heitmeyer. Formal Methods: A Panacea or Academic Poppycock? In *Proceedings of the 10th International Conference of Z Users (ZUM'97)*, LNCS 1212, pages 3–9. Springer, 1997.
 - [74] C.L. Heitmeyer. On the Need for Practical Formal Methods. In *Proceedings of the 5th International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, LNCS 1486, pages 18–26. Springer, 1998.
 - [75] C.L. Heitmeyer, M. Archer, E.I. Leonard, and J. McLean. Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*, pages 346–355, New York, NY, USA, 2006. ACM.
 - [76] C.L. Heitmeyer, M.M. Archer, E.I. Leonard, and J.D. McLean. Applying Formal Methods to a Certifiably Secure Software System. *IEEE Transactions on Software Engineering*, 34(1):82–98, 2008.
 - [77] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
 - [78] T. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal ACM*, 50(1):63–69, 2003.

-
- [79] X. Huang. Translating Machine-Generated Resolution Proofs into ND-Proofs at the Assertion Level. In *Proceedings of the of Pacific Rim International Conferences on Artificial Intelligence (PRICAI'96)*, LNCS 1114, pages 399–410. Springer, 1996.
 - [80] X. Huang and A. Fiedler. Proof Verbalization as an Application of NLG. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 965–970. Morgan Kaufmann, 1997.
 - [81] W. Hughes. *Critical Thinking*. Broadview Press, Petersborough, 1992.
 - [82] A. Hussey and B. Atchison. Safe Architectural Design Principles. Technical Report 00-19, Software Verification Research Centre, The University of Queensland, July 2000.
 - [83] M. Huth and M. Ryan. *Logic in Computer Science Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.
 - [84] C.B. Jones. Specification and Design of (Parallel) Programs. In *Proceedings of the 9th World Computer Congress Information Processing (IFIP'83)*, pages 321–332, 1983.
 - [85] N.D. Jones and A.J. Glenstrup. Program Generation, Termination, and Binding-time Analysis. In *Proceedings of the Generative Programming and Component Engineering (GPCE'02)*, LNCS 2487, pages 1–31. Springer, 2002.
 - [86] T.P. Kelly. *Arguing Safety - A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, 1998.
 - [87] T.P. Kelly. Reviewing Assurance Arguments – A Step-by-Step Approach. In *Proceedings of Workshop on Assurance Cases for Security - The Metrics Challenge, Dependable Systems and Networks*. IEEE, July 2007.
 - [88] T.P. Kelly and J.A. McDermid. Safety Case Patterns – Reusing Successful Arguments. In *Proceedings of IEE Colloquium on Understanding Patterns and Their Application to System Engineering*, Digest No. 1998/308, pages 3/1–3/9. Institute of Electrical Engineers, 1998.
 - [89] T.P. Kelly and R. Weaver. The Goal Structuring Notation - A Safety Argument Notation. In *Proceedings of the DSN Workshop on Assurance Cases: Best Practices, Possible Outcomes, and Future Opportunities*, Florence, Italy, July 2004.
 - [90] S. Kinnersly. Whole Airspace ATM System Safety Case: Preliminary Study. Technical report, EUROCONTROL, November 2001.
 - [91] C. Lacave and F. Diez. A Review of Explanation Methods for Bayesian Networks. Technical report, Dept. Inteligencia Artificial, UNAD, Madrid, 2000.

-
- [92] C. Lacave and F.J. Diez. A Review of Explanation Methods for Heuristic Expert Systems. *The Knowledge Engineering Review*, 19(2):133–146, 2004.
 - [93] T. Lelievre, J. Lapie, R. Beaulieu, and R. Rattier. AFI RVSM Programme Functional Hazard Assessment. Technical report, ALTRAN Technologies CNS/ATM Division, May 2005.
 - [94] N.G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
 - [95] N.G. Leveson. The Role of Software in Spacecraft Accidents. *AIAA Journal of Spacecraft and Rockets*, 41:564–575, 2004.
 - [96] N.G. Leveson, S.S. Cha, and T.J. Shimeall. Safety Verification of ADA Programs using Software Fault Trees. *IEEE Software*, 8(4):48–59, 1991.
 - [97] N.G. Leveson and P.R. Harvey. Analyzing Software Safety. *IEEE Transactions Software Engineering*, 9(5):569–579, 1983.
 - [98] N.G. Leveson and C.S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.
 - [99] Tube Lines Limited. Tubes Lines Contractual Safety Case. Technical report, May 2004.
 - [100] B. Littlewood and D. Wright. The Use of Multilegged Arguments to Increase Confidence in Safety Claims for Software-Based Systems: A Study Based on a BBN Analysis of an Idealized Example. *IEEE Transactions Software Engineering*, 33(5):347–365, 2007.
 - [101] W. McCune and O. Shumsky-Matlin. Ivy: A Preprocessor and Proof Checker for First-Order Logic. *Computer-Aided Reasoning: ACL2 Case Studies, Advances in Formal Methods*, 4:265–282, 2000.
 - [102] J. Moore. *Participating in Explanatory Dialogues: Interpreting and Responding to Questions in Context*. MIT Press, 1994.
 - [103] J. Moore and W. Swartout. Pointing: A Way Toward Explanation Dialogue. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 457–464, Boston, MA, 1990.
 - [104] B. Moulin, H. Irandoust, M. Belanger, and G. Desbordes. Explanation and Argumentation Capabilities: Towards the Creation of More Persuasive Agents. *Artificial Intelligence Review*, pages 169–222, 2002.
 - [105] J. Murdoch. *PSM Safety Measurement White Paper*, PSM Safety and Security TWG, January 2006.

-
- [106] G.C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, France, Jan 1997. ACM.
 - [107] G.C. Necula. Translation Validation for an Optimizing Compiler. *SIGPLAN Notes*, 35(5):83–94, 2000.
 - [108] C. O'Halloran. Issues for the Automatic Generation of Safety Critical Software. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 277–280, Washington, DC, USA, 2000. IEEE Computer Society.
 - [109] C. O'Halloran. Model Based Code Verification. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM'03)*, LNCS 2885, pages 16–25. Springer, 2003.
 - [110] P. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning About Programs that Alter Data Structures. In *Proceedings of Computer Science Logic*, LNCS 2142, pages 1–19. Springer, 2001.
 - [111] Department of Defence. *MIL-STD-882D Standard Practice For System Safety*, February 2000.
 - [112] Ministry of Defence. *00-56 Safety Management Requirements for Defence System*, June 2007.
 - [113] UK Ministry of Defence. *00-56 Safety Management Requirements for Defence Systems*, 2007.
 - [114] E. Ong. Fault Protection in a Component-Based Spacecraft Architecture. MSc thesis. Massachusetts Institute of Technology, 2003.
 - [115] New York Independent System Operator. NYISO Interim Report August 14, 2003 Blackout. 2004-01-08. Technical report, 2004.
 - [116] M. Ouimet. Formal Software Verification: Model Checking and Theorem Proving. Technical Report Embedded Systems Laboratory Technical Report ESL-TIK-00214, Massachusetts Institute of Technology.
 - [117] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6(4):319–340, 1976.
 - [118] D. Pastre. MUSCADET 2.3: A Knowledge-Based Theorem Prover Based on Natural Deduction. In *Proceedings International Joint Conference on Automated Reasoning (IJCAR'01)*, LNCS 2083, pages 685–689. Springer, 2001.
 - [119] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society, 1977.

-
- [120] A. Pnueli. System Specification and Refinement in Temporal Logic. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'92)*, LNCS 652, pages 1–38. Springer, 1992.
 - [121] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1384, pages 151–166. Springer Berlin, 1998.
 - [122] W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Do You Trust Your Model Checker? In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD '00)*, LNCS 1954, pages 179–196, London, UK, 2000. Springer.
 - [123] D.W. Reinhardt. Use of the C++ Programming Language in Safety Critical Systems. MSc thesis. University of York, 2004.
 - [124] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *ACM*, 12(1):23–41, 1965.
 - [125] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.
 - [126] J. Rushby. A Safety-Case Approach for Certifying Adaptive Systems. In: *AIAA Infotech@Aerospace Conference*, 2009.
 - [127] D. Sannella and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. *Acta Informatica*, 25:233–281, 1988.
 - [128] K. Schloegel, D. Oglesby, E. Engstrom, and D. Bhatt. Composable Code Generation for Model-Based Development. In *Proceedings of the Software and Compilers for Embedded Systems*, LNCS 2826, pages 211–225. Springer, 2003.
 - [129] M. Schroeder. Towards a Visualization of Arguing Agents. *Future Generation Computer Systems*, 17(1):15–26, 2000.
 - [130] A. Prasad Sistla. Safety, Liveness and Fairness in Temporal Logic. *Formal Aspects of Computing*, 6(5):495–512, 1999.
 - [131] D.R. Smith. (KIDS): A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.
 - [132] System Safety Society. *System Safety Analysis Handbook*. Federal Aviation Administration, 2nd edition, 1997.
 - [133] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, June 1992.

-
- [134] Z. Stachniak. Resolution Proof Systems with Weak Transformation Rules. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, pages 38–43, New York, NY, USA, 1990. ACM.
 - [135] Z. Stephenson, T.P. Kelly, and J-L. Camus. Developing an Argument for Def Stan 00-56 from Existing Qualification Evidence. To appear in *Proceedings of the Embedded Real-Time Software and Systems (ERTS²'10)*, Toulouse, France, 2010.
 - [136] M.E. Stickel, R.J. Waldinger, M.R. Lowry, T. Pressburger, and I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, pages 341–355, London, UK, 1994. Springer.
 - [137] L. Strigini. Formalism and Judgement in Assurance Cases. In *Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, Florence, Italy, 2004. IEEE.
 - [138] I. Stürmer and M. Conrad. Test Suite Design for Code Generation Tools. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, page 286, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
 - [139] I. Stürmer, D. Weinberg, and M. Conrad. Overview of Existing Safeguarding Techniques for Automatically Generated Code. In *Proceedings of the 2nd International ICSE Workshop on Software Engineering for Automotive Systems (SEAS'05)*, volume 30, pages 1–6, New York, NY, USA, 2005. ACM.
 - [140] G. Sutcliffe. Semantic Derivation Verification: Techniques and Implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.
 - [141] S.N. Thomas. *Practical Reasoning in Natural Language*. Prentice-Hall, 1981.
 - [142] S. Trac, Y. Puzis, and G. Sutcliffe. An Interactive Derivation Viewer. In *Proceedings of the 7th Workshop on Workshop on User Interfaces for Theorem Provers*, ENTCS 174, pages 109–123, 2006.
 - [143] A.C. Tribble, S.P. Miller, and D.L. Lempia. Software Safety Analysis of a Flight Guidance System. In *Proceedings of the 21st Digital Avionics Systems Conference (DASC'02)*, volume 2, pages 13C1–1–13C1–10, Irvine, California, October 2002.
 - [144] E.D. Vries, K. Lund, and M. Baker. Computer-Mediated Epistemic Dialogue: Explanation and Argumentation as Vehicle for Understanding Scientific Notions. *The Journal of Learning Sciences*, 11(1):63–103, 2002.
 - [145] J. Wang. Offshore Safety Case Approach and Formal Safety Assessment of Ships. *Journal of Safety Research*, 33(1):81–115, 2002.

-
- [146] R.A. Weaver. *The Safety of Software Constructing and Assuring Arguments*. PhD thesis, University of York, 2003.
 - [147] R.A. Weaver, J.A. McDermid, and T.P. Kelly. Software Safety Arguments: Towards a Systematic Categorisation of Evidence. In *Proceedings of the 20th International System Safety Conference*, Denver, USA, 2002.
 - [148] K.A. Weiss. Component-Based Systems Engineering for Autonomous Spacecraft. MSc thesis. Massachusetts Institute of Technology, 2003.
 - [149] M. Wenzel. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, LNCS 1690, pages 167–184. Springer, 1999.
 - [150] M. Wenzel. *Isabelle/Isar - A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.
 - [151] M.W. Whalen and M.P.E. Heimdahl. An Approach to Automatic Code Generation for Safety-Critical Systems. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 315–318, Cocoa Beach, FL, USA, Oct 1999. IEEE Computer Society.
 - [152] M.W. Whalen and M.P.E. Heimdahl. On the Requirements of High-Integrity Code Generation. In *Proceedings of the Fourth High Assurance in Systems Engineering Workshop, 4th IEEE International Symposium*, pages 217–224, Washington DC, 1999.
 - [153] J. Whittle and J. Schumann. Automating the Implementation of Kalman Filter Algorithms. *ACM Transactions on Mathematical Software*, 30(4):434–453, 2004.
 - [154] J. Wielemaker. *SWI-Prolog 5.2.9 Reference Manual*. Amsterdam, 2003.
 - [155] S. Wilson, T.P. Kelly, and J.A. McDermid. Safety Case Development: Current Practice, Future Prospects. In *Safety and Reliability of Software Based Systems - Twelfth Annual CSR Workshop*, Bruges, Belgium, 1997. Springer.
 - [156] W. Wong. Validation of HOL Proofs by Proof Checking. *Formal Methods in System Design: An International Journal*, 14(2):193–212, 1999.
 - [157] W. Wu. *Architectural Reasoning for Safety-Critical Software Applications*. PhD thesis, University of York, 2007.
 - [158] F. Ye. *Justifying the Use of COTS Components within Safety Critical Applications*. PhD thesis, University of York, 2005.
 - [159] L.D. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A Translation Validator for Optimizing Compilers. ENTCS 65, pages 2–18. Elsevier, 2002.