A framework for explicit modelling of domain knowledge in state-based formal methods : the case of interactive critical systems

Ismail Mendil

To cite this version:

Ismail Mendil. A framework for explicit modelling of domain knowledge in state-based formal methods : the case of interactive critical systems. Computer science. Institut National Polytechnique de Toulouse - INPT, 2023. English. NNT : 2023INPT0074 . tel-04611765

HAL Id: tel-04611765
https://theses.hal.science/tel-04611765v1

Submitted on 14 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

M. ISMAIL MENDIL

le jeudi 5 octobre 2023

Titre :

Un cadre formel pour la modélisation explicite des connaissances de domaine dans les méthodes formelles basées états : le cas des systèmes critiques interactifs.

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. YAMINE AIT AMEUR
M. PHILIPPE PALANQUE

Rapporteurs :

M. JEAN VANDERDONCKT, UNIVERSITE CATHOLIQUE DE LOUVAIN
M. MARC FRAPPIER, UNIVERSITE DE SHERBROOKE
MME OLGA KOUCHNARENKO, UNIVERSITE DE FRANCHE COMTE

Membre(s) du jury :

MME KATHIA MARCAL DE OLIVEIRA, UNIVERSITE DE VALENCIENNES, Président
M. DOMINIQUE MERY, UNIVERSITÉ LORRAINE, Membre
M. FUYUKI ISHIKAWA, NATIONAL INSTITUTE OF INFORMATICS TOKYO, Invité(e)
M. NEERAJ SINGH, TOULOUSE INP, Membre
M. PHILIPPE PALANQUE, UNIVERSITE TOULOUSE 3, Membre
M. YAMINE AIT AMEUR, TOULOUSE INP, Membre

to my parents and my brothers,

Abstract

System engineering advocates an explicit modelling of domain knowledge at early stages of the development cycle. Moreover, integrating contextual information and certification standard requirements into formal models enhances their quality and reliability. On the one hand formal methods provide primitives for modelling components and views of these systems but they are not endowed with built-in primitives for explicit modelling contextual constraints, and more broadly, domain knowledge associated with these formal models. Consequently relevant domain knowledge is implicitly hardcoded in the system formal specification or is, in the worse case, overlooked. On the other hand ontologies have demonstrated their efficiency in modelling domain-specific features but they are not available as built-in primitives in formal methods.

The goal of this thesis is to propose a framework and associated methodology for modelling explicitly domain knowledge in formal modelling. As a byproduct, consequences of the explicit modelling of domain knowledge are investigated including formal standard-based conformance checking of interactive critical systems and domain-specific behavioural analyses of formal models. In our research effort, we defined an integrated framework for addressing the explicit modelling of domain knowledge problem in formal modelling. The framework specified three main steps: formalising domain knowledge, annotating formal models and transferring domain knowledge constraints. The formalisation is achieved by proposing an ontology modelling language in the form of a generic Event-B theory, the annotation consists in typing elements of the model with the concepts of an ontology and the transferring of predefined domain properties is supported by the description of a methodological rule of using exclusively the data types and operators of the Event-B theory (closure). Furthermore, several cases studies from the interactive critical systems realm have been addressed to showcase the generality, effectiveness and advantages of the framework. First, conformance checking is a prominent consequence of explicit modelling of domain knowledge; standard conformance to ARINC 661 certification standard of a cockpit application user interface is addressed as a special case of explicit modelling of domain knowledge. A second consequence of the framework is the definition of a method and formal Event-B theories for specifying domain-specific behavioural analyses.

Résumé

L'ingénierie système préconise une modélisation explicite des connaissances de domaine aux premières étapes du cycle de développement. De plus, L'intégration d'informations contextuelles et d'exigences provenant de normes de certification dans des modèles formels améliore leur qualité et leur fiabilité. D'une part, les méthodes formelles fournissent des primitives d'abstraction pour modéliser la structure et les comportements de ces systèmes, mais elles ne sont pas équipées pas de primitives spécifiquement dédiées pour modéliser explicitement les contraintes contextuelles, et plus largement, les connaissances de domaine associées à ces modèles formels. Par conséquent, les connaissances de domaine sont implicitement codées en dur dans la spécification formelle du système où sont, dans le pire des cas, ignorées. D'autre part, les ontologies ont démontré leur efficacité dans la modélisation des connaissances, mais elles ne sont pas fournies en tant que primitives dans les méthodes formelles.

L'objectif de cette thèse est de proposer un cadre et une méthodologie associée pour modéliser explicitement les connaissances de domaine dans le contexte la modélisation formelle. Par ailleurs, les conséquences de la modélisation explicite des connaissances de domaine sont étudiées, y compris la vérification formelle de la conformité par rapport aux normes des systèmes critiques interactifs ainsi que les analyses comportementales spécifiques au domaine appliquées à des modèles formels. Dans notre effort de recherche, nous avons défini un cadre intégré pour résoudre le problème de la modélisation explicite des connaissances de domaine dans la modélisation formelle. Le cadre a spécifié trois étapes principales, à savoir formaliser les connaissances de domaine, annoter les modèles formels et enfin transférer les propriétés des connaissances de domaine. La formalisation est réalisée en proposant un langage de modélisation à base d'ontologies sous forme de théorie Event-B générique, l'annotation consiste à typer les éléments du modèle avec les concepts d'une ontologie de domaine et le transfert de propriétés de domaine prédéfinies est régi par la description d'une règle méthodologique stipulant l'utilisation exclusive des types de données et des opérateurs de la théorie Event-B formalisant une ontologie de domaine. En outre, plusieurs études de cas fournies dans le domaine des systèmes critiques interactifs sont abordées pour montrer la généralité, l'efficacité et les avantages du cadre. Premièrement, la vérification de conformité est une conséquence importante de la modélisation explicite des connaissances de domaine dans la modélisation formelle ; la conformité à la norme de certification ARINC

661 d'un système interactif d'une application de cockpit est utilisée pour démontrer l'efficacité du cadre. Une deuxième conséquence du cadre est la définition d'une méthode et de théories d'Event-B pour spécifier des analyses comportementales spécifiques à un domaine. En outre, une étude de cas concrète est décrite et analysée pour illustrer la méthode conçue.

Acknowledgements

This research endeavor and outcome would not have been possible without Yamine Ait Ameur, professor at Toulouse INP-ENSSEIHT/IRIT, Neeraj Kumar Singh, associate professor at Toulouse INP-ENSSEIHT/IRIT, Dominique Méry, professor at Université de Lorraine-Telecom Nancy/LORIA and Philippe Palanque, professor at Université Toulouse III - Paul Sabatier/IRIT. I would like to express my deepest gratitude to them for their trust, scientific advice, their support in the hard times, and kindness in the good times.

I'm extremely grateful to Kathia Oliveira, professor at Université Polytechnique Hauts-de-France, who presided over my defense. I am deeply indebted to Olga Kouchnarenko, professor at Université de Franche-Comté, Jean Vanderdonckt, professor at Université Catholique de Louvain and Marc Frappier, professor at Université de Sherbrooke who took the time to review this thesis, provided me with invaluable feedback and positive suggestions. I would like to express my gratitude to Fuyuki Ishikawa, associate professor at the national institute of informatics, who accepted to be part of my thesis defense committee.

I express my acknowledgement to Toulouse INP-ENSEEIHT and IRIT for the support and the funding which allowed me to pursue my scientific research in Toulouse. Furthermore, I am grateful to them for organising the opportunity of going abroad and of pursuing a doctoral internship at National Institute of Informatics (NII) at Tokyo. Alongside, I would like to thank Fuyuki Ishikawa and Tsutomu Kobayashi, researcher at Japan Aerospace Exploration Agency (JAXA), Yamine Ait Ameur and Neeraj Kumar Singh for their trust, priceless supervision, and kindness during my internship at NII. My two-month-long internship was funded by the NII.

This thesis was funded by the FORMEDICIS (FORmal MEthods for the Development and the engineering of Critical Interactive Systems) ANR-16-CE25-0007 and EBRP (EventB-Rodin-Plus) ANR-19-CE25-0010.

I thank the people at Toulouse INP-ENSEEIHT, IRIT and ACADIE team with whom I enjoyed 4 years of learning, sharing and kindness: Yamine, Neeraj, Peter, Nassima, Mohamed, Guillaume, Marc, Xavier, Sarah, Yannis, Benoît, Jonathan, , Nesrine, Aurélie, Philippe Q. and Philippe M.. Moreover, I would like to thank Annabelle, Sylvie, Vanessa, Léna and Sandrine for their wonderful support in my administrative paperwork and for their kindness.

Last and not the least, my sincere thanks go to my family and friends for their support: Amine, Dalila, Hamid, Lotfi, Meriem, Mohamed, Naim and Said.

Contents

Introduction **1**

I Background 9

1 Event-B: a Correct-by-Construction Method **11**

1.1 State-Based Formal Modelling 12
1.2 Proof-Based Verification 14
1.3 Event-B Theories Extension 15
1.3.1 Extending Event-B Language with Theories 15
1.3.2 Event-B Theories Structure 16
1.3.3 Well-Definedness 18
1.3.4 Importing Theories 19
1.4 IDE: Rodin Platform and Plug-ins 19

2 Domain Knowledge in Formal Modelling **21**

2.1 Modelling Domain Knowledge 22
2.1.1 On the Importance of Domain Knowledge 22
2.1.2 On the Lack of a Generic Approach 23
2.1.3 Ontologies as Domain Knowledge Model 24
2.2 The Ontology Formalism 25
2.2.1 Fundamental Characteristics 27
2.2.2 Semantic Annotation Using Ontologies 28
2.2.3 Ontologies for Engineering Contexts 28
2.3 Standards as Domain Knowledge 29
2.4 Synthesis and Conclusion 31

3 Interactive Critical Systems **33**

3.1 Formal Methods for Interactive Systems 34
3.1.1 Interactive Systems Characteristics 34
3.1.2 Formal Design of Interactive Critical Systems 36
3.2 Interactive Systems Development 39
3.3 The Context of The FORMEDICIS project 42
3.4 Synthesis and Conclusion 42

4 Case studies **45**
4.1 Traffic Collision Avoidance System 45
4.1.1 Overview of Operation 46
4.1.2 Definitions and Requirements 46
4.2 Multi-Purpose Interactive Application 48
4.2.1 Requirements of WXR User Interface 49
4.3 Automatic Teller Machine 50
4.4 Conclusion 51

II Contributions **53**

The Road Map of the Contributions **55**
Explicit Modelling of Domain Knowledge 57
Transferring of Safety Properties 57
Analysis of Behavioural Properties 58
Formal Conformance Checking 58

5 Explicit Modelling of Domain Knowledge Using Ontologies **61**
5.1 Temperature Aggregator Example 62
5.1.1 Temperature Aggregator Requirements 62
5.1.2 Modelling without the Theory Operators 63
5.1.3 Modelling with the Theory Operators 64
5.1.4 Synthesis 67
5.2 An Ontology Modelling Language (OML) 68
5.2.1 OML as a Generic Event-B Theory 69
5.2.2 `OntologiesTheory` - Data type 69
5.2.3 `OntologiesTheory` - Operators 69
5.2.4 `OntologiesTheory` - Theorems 73
5.3 Conclusion 73

6 Annotation-Based Transfer of Safety Properties **75**
6.1 Our Approach 76
6.1.1 Generic Part: the definition of the Domain Theory 77
6.1.2 Specific Part: Annotating the System 77
6.2 TCAS Case Study 77
6.2.1 An Ontology of Interactive Objects 78
6.2.2 Instantiation of the Displayability Theory 84
6.2.3 Modelling without the Theory Operators 85
6.2.4 Modelling with the Theory Operators 87
6.2.5 Proof Statistics 89
6.3 Conclusion 90

7 Annotation-Based Analysis of Behavioural Properties **91**
7.1 Our Approach . . . 92
7.2 The Event-B Meta-Theory . . . 93
7.2.1 Event-B Machine Structure as a Data Type . . . 93
7.2.2 Event-B Machine Proof Obligations as Predicates . . . 95
7.2.3 Modelling with Event-B Meta-Theory . . . 96
7.3 A Framework for Behavioural Analyses . . . 96
7.3.1 The Architecture of the Framework . . . 97
7.3.2 How does the Framework Work? . . . 98
7.4 The Framework at Work . . . 100
7.4.1 Defining a Domain-Specific Behavioural Analysis . . . 100
7.4.2 Applying a Domain-Specific Behavioural Analysis . . . 106
7.5 Advantages of the Framework . . . 111
7.5.1 Principled Approach and Reusability . . . 111
7.5.2 Non-intrusiveness . . . 111
7.5.3 Verification Based on Theorem Proving . . . 111
7.5.4 Proof and Modelling Effort Reduction . . . 112
7.5.5 Generalisation . . . 113
7.6 Conclusion . . . 113

8 Formal Conformance Checking **115**
8.1 Introduction . . . 116
8.2 Our approach . . . 116
8.2.1 A Standard Formal Specification —(2) on Figure 8.1 . . . 117
8.2.2 Standard Theory Instantiation —(3) on Figure 8.1 . . . 118
8.2.3 Model Annotation —(4) on Figure 8.1 . . . 119
8.3 Formalisation of ARINC 661 Standard . . . 119
8.3.1 `ARINC661Theory` - Concepts Declaration . . . 120
8.3.2 `ARINC661Theory` - Operators Declaration . . . 121
8.3.3 `ARINC661Theory` - Primitives Definitions . . . 122
8.3.4 `ARINC661Theory` - Theorems . . . 124
8.4 Weather Radar Application Case Study . . . 124
8.4.1 `WXRTheory` - Instances Declaration . . . 125
8.4.2 `WXRTheory` - Instances Definition . . . 125
8.4.3 `WXRTheory` - Operators Declaration and Definition . . . 126
8.4.4 `WXRTheory` - Theorems . . . 127
8.4.5 Annotated Model of WXR —(4) on Figure 8.1 . . . 128
8.5 Advantages of The Framework . . . 130
8.5.1 Achieving Standard Conformance Formally . . . 130
8.5.2 Qualitatively Enhanced System Models . . . 131
8.5.3 Reduction of Modelling and Proving Effort . . . 131
8.5.4 Enabling Evolution of Standard . . . 132
8.6 Conclusion . . . 132

Conclusion and Perspectives **132**

Bibliography **137**

Appendices **159**

A Meta-Modelling Theories **161**
A.1 OntologiesTheory - Ontology Modelling Language 162
A.2 EvtBTheo - Event-B Meta-Theory 163

B Domain Theories **165**
B.1 DisplayabilityTheory - Displayability Domain Theory 166
B.2 ARINC661Theory - ARINC 661 Standard Domain theory 167
B.3 Domain-Specific Behaviour Analysis 168
B.3.1 BehaviouralPropertiesTheory - Analysis Operator . . . 168
B.3.2 Theo4Reachability - Analysis Low-Level Terms 168
B.3.3 EvtBManip - Auxiliary Operators 168

C Case Studies **169**
C.1 Temperature Aggregator Case Study Modelling 170
C.1.1 C_TemperatureContext - Event-B Context for Units . . . 170
C.1.2 C_TemperatureMachine - Machine without Operators . . 170
C.1.3 ThermalUnits - Event-B Theory for Units 170
C.1.4 T_TemperatureMachine - Event-B Machine with Operators 170
C.2 TCAS Case Study Modelling 171
C.2.1 InstantiationContext - Event-B Context for Instantiation 171
C.2.2 SetTheoriticOperationsBasedModel 171
C.2.3 TheoryOperatorsBasedModel 171
C.3 ATM Case Study Modelling 172
C.3.1 ATMEnvironment - Event-B Context for Constants 172
C.3.2 ATMUserInterface - Event-B Machine for ATM Model . 172
C.3.3 ATMmEBModel - ATM Model Exported to EB4EB 172
C.3.4 AnnotatedModel - Annotation and Analysis 172
C.4 WXR Case Study Modelling 173
C.4.1 WXRModel - Event-B Machine for WXR Model 173

List of Figures

3.1 MVC architecture model [104] 40

4.1 TCAS Protection Volume 46
4.2 Standardized Symbology for Use on the Traffic Display 47
4.3 Snapshots of MPIA 49
4.4 A map of the contributions of the thesis 56

6.1 The framework for modelling with explicit domain knowledge . . 76

7.1 A framework for domain-specific behavioural analyses 93
7.2 Fine-grain view of the behavioural analysis framework 98
7.3 A tree representation of an ontology of events 101
7.4 Simplified depiction of the necessary reachability analysis 104

8.1 The framework for standard conformance-by-construction 118
8.2 WXR system annotated with ARINC 661 concepts 129
8.3 A map of thesis contributions and perspective 135

List of Tables

1.1 Event-B machine proof obligations 14
1.2 Event-B refinement proof obligations 15

6.1 Proof statistics of OML and TCAS case study 89

7.1 Proof statistics of behavioural analysis and ATM case study . . . 112

8.1 Mapping between ARINC 661 concepts and Event-B formalisation 120
8.2 Proof statistics of conformance checking and MPIA case study . 131

Introduction

The development of critical systems needs to address safety requirements to guarantee that the system does not fall in unsafe, undesirable or dangerous states. This verification is critical when such errors may jeopardise human lives, the environment, or have negative economic or societal effects. In the life cycle of such critical system, formal methods have demonstrated their significant effectiveness in preventing bad design decisions from affecting the specification and implementation of systems at different development and deployment stages. In this context, environment assumptions and domain-specific requirements are essential to improve confidence in the models of these critical systems. Furthermore, domain-specific requirements contribute more to enhancing the quality of system models when taken into account at the design and specification stages of the system engineering life cycle. Indeed, formal system modelling with explicit modelling of domain knowledge and evolving contexts and system environment is at the centre of the work presented in this thesis.

Problem Statement

THe FORMEDICIS [1] ANR projet aimed at designing domain-specific formal modelling language called FLUID, dedicated to formal modelling and verification of interactive critical systems. In this context, several domain properties related to interactive critical systems have been identified such as *every input shall be followed by a confirmation*. Indeed, interactive critical systems represent a starting point as the source of domain knowledge properties and case studies for supporting the definition of a framework of explicit modelling of domain knowledge.

The seminal work of [171] and [29, 32] advocates the separation of so called *Domain Knowledge* from the system specification, and proposes the well-known triptych $K, S \vdash R$, where K represents the domain-specific knowledge made of concepts and properties; S represents a system model; and R represents the expressed system requirements. This separation is motivated by two main arguments: first, domain knowledge is usually stable and reusable, and second, its formalisation is made explicit through K. Last, the relationship between

[1] https://anr.fr/Projet-ANR-16-CE25-0007

these three features, symbolised by $\vdash$, states that the domain knowledge and system description entail the systems desired properties.

However, on the one hand, domain knowledge is essential to prove system safety but it is often implicitly modelled, i.e. the domain-specific requirements are usually hard encoded in the system formal model, or overlooked, remaining informal in the system's documentation or in the mind of the designers[14, 152]. This way of formal modelling is at best inefficient and at worst may lead to unverifiable critical properties of these systems. First, it is inefficient because common domain-specific requirements are formalised for each system in an ad hoc way, yielding a poor methodology of specification in terms of reuse and sharing. Second, it may lead to incomplete formal specification if the domain-specific requirements are implicitly assumed. On the other hand, system engineering approaches, particularly formal methods do not offer specific stable and consensual constructs for explicit modelling of domain knowledge, although many notations and modelling languages have been proposed therefore leading to various and heterogeneous domain knowledge expression. Nevertheless, there exist formal modelling languages and/or meta-models, sometimes standardised [40] supporting the formalisation of domain knowledge. Transformations are often required to reuse domain knowledge formalisations in the setup of formal methods. As a result, heterogeneous formalisations are produced, compromising sharing, and reuse.

In order to address above mentioned issues, there is a need for resources and frameworks to systematically support *explicit modelling of domain knowledge in formal modelling* where domain-specific safety and behavioural properties may be systematically formalised and used by system formal models.

Objectives of this thesis

In the context of this thesis, the purpose of our work was *to provide a framework and resources for explicit modelling of domain knowledge in the formal modelling of systems*. In particular, we have considered explicit modelling of domain knowledge in the formal modelling of interactive critical systems. Why? Addressing this difficulty would undoubtedly enable the construction of higher-quality formal specifications and significantly reduces verification effort on the system-specific modeling front. The separation of concerns between domain-specific shared requirements and system-specific special requirements shall be allowed by the framework to enhance understandability and traceability of the formal specification of systems. Additionally, the verification process becomes more efficient because the common properties are established only once to be theorems of the domain-specific part. Therefore, the framework will foster the reuse and the sharing of formal specifications of requirements and their verification. Interactive critical systems fed and supported the definition of the framework. This choice has been motivated by the domain-specific properties and case studies featuring a neat distinction between domain-specific and system-specific properties.

The formal methods addressed in this thesis are state-based and refinement-based. In particular, Event-B formal method has been preferred to implement the framework. Why? Event-B is an excellent choice for systems formal specification and verification because (1) it supports refinement-based correct-by-construction development, (2) its specification language relies on the expressive set theory and first-order logic, (3) the theorem proving is supported for verification thus it does not suffer from the limitations inherent to model checking techniques. A decisive factor determining the selection of Event-B is the *Event-B extensions mechanism supported by the Theory Plug-in*; these extensions are called *Event-B theories*. Indeed, Event-B theories support the definition of generic specifications allowing contract-based modelling relying on the assume-guarantee paradigm.

Contributions

In the endeavour to meet the objective stated previously, our research resulted in various contributions. Our contributions are summarised below.

Explicit Modelling of Domain Knowledge Using Ontologies A framework for explicit modelling of domain knowledge was proposed and illustrated through several case studies. Indeed, the motivation why explicit modelling of domain knowledge is discussed, additionally an illustrative and didactic model is developed to explain the differences of explicit modelling of domain knowledge in contrast with its implicit modelling. Next, an important building block of the framework, our ontology modelling language as generic Event-B theory, is developed and the reasons of the necessity of such language are explained.

Annotation-Based Transferring of Properties The framework for explicit modelling of domain knowledge has been used to demonstrate a methodology for transferring domain-specific properties to system models. Two kinds of annotations were defined and handled: (1) annotation of state variables and (2) annotation of events. (1) the typing of the system state variables with domain ontology concepts is devised as annotation mechanism. This annotation mechanism allows the reuse and transfer of domain safety properties to system models. In particular, the TCAS case study has been refactored to illustrate the methodology. (2) A annotation-based methodology for defining and applying domain-specific behavioural analyses to system models is proposed. This methodology is based on the formalisation of domain-specific behavioural analyses using the ontology modelling language and the reflexive Event-B framework (EB4EB) [143]. The annotation we defined links the domain concepts to the model's events allowing the application of a behavioural analysis. In particular, the methodology has been exemplified using the ATM case study.

Formal Conformance Checking We demonstrated that conformance checking to standards may be regarded as a special case of explicit modelling of domain

knowledge and safety property transferring. For this purpose, a methodology for formalising standards specifications has been defined, and a sufficient rule of construction has been explained to guarantee the checking of the conformance to the standard. Therefore, this approach to conformance checking is coined conformance by construction. To showcase the methodology, the ARINC 661 standard is formalised as an Event-B theory using the ontology modelling language and the WXR case study has been modelled to illustrate the methodology.

Structure of this thesis

The manuscript is composed of two parts: Part I - Background and Part II - Contributions. The first part is composed of 4 chapters. First, chapter 1 gives an overview on Event-B method. Next, chapter 2 reviews the work related to domain knowledge modelling in formal methods where the lack of framework for explicit modelling of domain knowledge is highlighted. Then, the importance of ontology formalism for knowledge representation is explained., and chapter 3 studies the field of formal modelling of interactive critical systems which fed and supported the definition of the framework of explicit modelling of domain knowledge. Last, chapter 4 describes the case studies used throughout this thesis to demonstrate the effectiveness and efficiency of the proposed framework.

The second part discusses the contributions of this thesis. It contains a roadmap and 4 chapters. The raodmap chapter draws an overview of the work carried out during the thesis, depicts a graphical representation of the interconnections of the contributions for tackling the challenge of explicit modelling of domain knowledge, and links the different contributions to the observations formulated in the part I - Background. Chapter 5 explains the importance of explicit modelling of domain knowledge and presents the ontology modelling language. The contribution *Annotation-Based Transferring of Properties* is discussed in two chapters. First, chapter 6 discusses the methodology to transfer safety properties specified in a domain ontology to system models. Chapter 7 presents the methodology to define and apply annotation-based domain-specific behavioural analyses to system models. Last, chapter 8 demonstrates that conformance checking may be achieved as a special case of explicit modelling of domain knowledge and domain-specific property transferring.

Last, a conclusion and perspectives chapter ends this manuscript.

Associated Projects

This thesis was undertaken as part of the FORMEDICIS (FORmal MEthods for the Development and the engineering of Critical Interactive Systems) ANR-16-CE25-0007 and EBRP (EventB-Rodin-Plus) ANR-19-CE25-0010.

Related Publications

Journal Articles

[115] **Ismail Mendil**, Yamine Aït-Ameur, Neeraj Kumar Singh, Guillaume Dupont, Dominique Méry and Philippe A. Palanque. Formal domain-driven system development in Event-B: Application to interactive critical systems. *Journal of Systems Architecture: Embedded Software Design (JSA)*, 135:102798, 2023. doi:10.1016/j.sysarc.2022.102798

[151] Neeraj Kumar Singh, Yamine Aït-Ameur, **Ismail Mendil**, Dominique Méry, David Navarre, Philippe A. Palanque, and Marc Pantel. F3FLUID: A formal framework for developing safety- critical interactive systems in FLUID. *Journal of Software: Evolution and Process*, page e2439, 2022. doi:10.1002/smr.2439

Conference Articles

[118] **Ismail Mendil**, Peter Rivière, Yamine Aït-Ameur, Neeraj Kumar Singh, Dominique Méry and Philippe A. Palanque. Non-Intrusive Annotation-Based Domain-Specific Analysis to Certify Event-B Models Behaviours. *In 29th Asia-Pacific Software Engineering Conference, APSEC 2022, Virtual Event, Japan, December 6-9, 2022*, pages 129–138. IEEE, 2022. doi:10.1109/APSEC57359.2022.00025.

[13] Yamine Aït-Ameur, Guillaume Dupont, **Ismail Mendil**, Dominique Méry, Marc Pantel, Peter Rivière and Neeraj Kumar Singh. mpowering the Event-B Method Using External Theories. In M. H. ter Beek and R. Monahan, editors, *Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano,Switzerland, June 7-10, 2022, Proceedings*, volume 13274 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2022. doi:10.1007/978-3-031-07727-2_2.

[116] **Ismail Mendil**, Yamine Aït-Ameur, Neeraj Kumar Singh, Dominique Méry and Philippe A. Palanque. Leveraging Event-B Theories for Handling Domain Knowledge in Design Models. In S. Qin, J. Woodcock, and W. Zhang, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 7th International Symposium, SETTA 2021, Beijing, China, November 25-27, 2021, Proceedings,* volume 13071 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2021. doi:10.1007/978-3-030-91265-9_3

[117] **Ismail Mendil**, Yamine Aït-Ameur, Neeraj Kumar Singh, Dominique Méry and Philippe A. Palanque. Standard Conformance-by-Construction with Event-B. *In A. Lluch-Lafuente and A. Mavridou, editors, Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris,*

France, August 24-26, 2021, Proceedings, volume 12863 of *Lecture Notes in Computer Science, pages 126–146. Springer, 2021.* doi:10.1007/978-3-030-85248-1_8.

[119] **Ismail Mendil**, Neeraj Kumar Singh, Yamine Aït-Ameur, Dominique Méry and Philippe A. Palanque. An Integrated Framework for the Formal Analysis of Critical Interactive Systems. *In 27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020*, pages 139–148. IEEE, 2020. doi:10.1109/APSEC51365.2020.00022.

[114] **Ismail Mendil**. A framework for critical interactive system formal modelling and analysis (A. Raschke, D. Méry, & F. Houdek, Eds.). *In: Rigorous state-based methods - 7th international conference, ABZ 2020, ulm, germany, may 27- 29, 2020, proceedings (A. Raschke, D. Méry, & F. Houdek, Eds.). Ed. by Raschke, A., Méry, D., & Houdek, F. 12071. Lecture Notes in Computer Science. Springer, 2020, 423–42*6. doi:10.1007/978-3-030-48077-6_36

Part I

Background

Chapter 1

Event-B: a Correct-by-Construction Method

This Chapter contains:

1.1 State-Based Formal Modelling 12
1.2 Proof-Based Verification 14
1.3 Event-B Theories Extension 15
  1.3.1 Extending Event-B Language with Theories 15
  1.3.2 Event-B Theories Structure 16
  1.3.3 Well-Definedness 18
  1.3.4 Importing Theories 19
1.4 IDE: Rodin Platform and Plug-ins 19

This chapter presents Event-B method since it is used to support the framework for explicit modelling of domain knowledge in formal modelling presented in PartII. Moreover, Event-B theories are extremely important since they allow to define generic formal specifications composed essentially of data types and operators that can be imported into the formal specification of systems composed essentially of contexts and machines. Therefore they enable a higher degree of reuse and sharing. However, for an advanced treatise of the Event-B formal method and further technical detail, the reader may use the reference book [3] as well as the website dedicated to Event-B method and its toolset [1].

[1] http://www.event-b.org/

<table>
<tr>
<td>
CONTEXT
  $ctxt_id$
SETS
  s
CONSTANTS
  c
AXIOMS
  A
THEOREMS
  T_{ctx}
END
</td>
<td>
MACHINE $machine_id$
SEES $ctxt_id$
VARIABLES
  x
INVARIANTS
  $I(x)$
  $T_{mach}(x)$

VARIANT
   $V(x)$
EVENTS
  EVENT evt
    ANY α
    WHERE $G(x, \alpha)$
    THEN
     $x :| BAP(x, \alpha, x^{'})$
    END
...
END
</td>
</tr>
</table>

Listings 1.1: Event-B model's structure: context & machine

1.1 State-Based Formal Modelling

Event-B is a formal method for modelling and verifying systems at early stages of the development life cycle. Therefore, it is useful for eliminating design errors, which are hard to find and expensive to correct in the implementations. Furthermore, Event-B is a state-based method whose language relies on typed set theory and first-order predicate logic.

Modelling in Event-B consists in formalising informal requirements of a system under design as a series of refinements of an initial abstract model (specification) leading to a final concrete model (implementation). This approach adopted by Event-B is accompanied with discharging proof-obligations to ensure the correctness of refinement of the initial specification. Therefore, the correctness of the implementation is ensured *by construction* thanks to the refinement.

A classical Event-B model contains mainly two kinds of components (see Listing 1.1): *contexts* and *machines*. First, the `CONTEXT` component describes the static part of a Event-B model. It contains the definitions, axioms and theorems needed to formalise the environment of the system and information that does not change when the system evolves. The main elements of a context are carrier sets `s`, constants `c`, axioms `A` and theorems $\mathtt{T}_{ctx}$.

Second, the `MACHINE` component describes the dynamic part of a model as a transition system. It has a set of events modifying a set of variables (the state of the machine) whose semantics is transition systems. A key operation allowing

modifying of a state is called `BAP` (*before-after predicate*). This predicate transformer specifies a substitution for a current state as well as for all the possible next states. Mathematically, the BAP is defined as

$$x :| BAP(x, \alpha, x') \tag{1.1}$$

where `BAP(x,` α`, x')` is a first-order formula linking the current state `x` and next state `x'` with parameters α. Besides, the events have guards `G(x,`α`)` conditioning the observation of the events.

Event-B machines may reference several contexts by enumerating these contexts in the `SEE` clause. Consequently, all sets, constants, axioms and the theorems are available for the machine.

Safety properties of the system may be specified as invariants or theorems of the machine. The clause `INVARIANTS` may contain `I(x)` and $\mathtt{T}_{mach}(x)$ which shall be proved to ensure that the machine establishes and preserves the specified properties. The difference between invariants and theorems lies in the proving status; invariants are proved inductively for each event of the machine, and the theorems are proved deductively using only the invariants and the context's axioms and theorems. The `VARIANT` clause appears in a machine containing some convergent events. This clause allows to describe reachability properties.

MACHINE *machine_id_2*
REFINES *machine_id_1*
VARIABLES
  x^C
INVARIANTS
  $J(x^A, x^C) \cdots$
EVENTS
  EVENT evt^C **REFINES** evt^A
    ANY α^C
    WHERE $G^C(x^C, \alpha^C)$
    WITH
      $x^{A\prime}, \alpha^A$: $W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$
    THEN
      $v^C :| BAP^C(x^C, \alpha^C, x^{C\prime})$
    END
$\ldots$

Listings 1.2: Machine refinement structure

A Machine may be refined by several concrete machines where the machine refined is specified in the `REFINES` clause. The concrete machine introduces a set of variables. However, variables of the refined machine (if any) can occur in an invariant. When it is the case, this invariant is said to be a gluing invariant; as this indicates, it "glues" the state space of the concrete machine to that of

the refined machine. Additionally, several events of a concrete machine may refine a single event of the refined machine. The refinement process may entail the generation of a number of proof obligations to ensure its correctness. The verification of the correctness of refinement process is discussed in Section 1.2.

Listing 1.2 shows an extract of a concrete machine with concrete variables, x^C, a gluing invariant $J(x^A, x^C)$ relating abstract and concrete variables and a refined event evt^C. Event guards (G^C) may be strengthened in order to model concrete system behaviour. Witnesses (`WITH` clause) for the parameter α^A of the abstract event evt^A and for abstract variables x^A may be given (predicate W). They are used for proving refinement correctness.

Notation. The superscripts ^{A} and ^{C} refer to abstract and concrete machines respectively.

1.2 Proof-Based Verification

Formal verification of Event-B models is achieved by discharging proof obligations that are automatically generated. For instance, consistency checking of an Event-B model with respect to formalised safety property is carried out by proving invariants preservation proof obligation (INV) for the Even-B machine.

(1) Theorems	$A \Rightarrow T_{ctx}$ $A \wedge I(x) \Rightarrow T_{mac}(x)$
(2) Invariant preservation (INV)	$A \wedge I(x) \wedge G(x, \alpha)$ $\wedge BAP(x, \alpha, x^{'}) \Rightarrow I(x')$
(3) Event feasibility (FIS)	$A \wedge I(x) \wedge G(x, \alpha)$ $\Rightarrow \exists x^{'} \cdot BAP(x, \alpha, x')$
(4) Variant progress	$A \wedge I(x) \wedge G(x, \alpha)$ $\wedge BAP(x, \alpha, x') \Rightarrow V(x') < V(x)$

Table 1.1: Event-B machine proof obligations

The proof obligations (PO) in Event-B are generated automatically for establishing the safety properties, reachability properties, the feasibility of transitions and the validity of a refinement. The main PO are listed in Listing 1.1. The prime notation, allowing us to define Before-After Predicates (BAP), denotes the next value of a variable after an event is observed. PO of Listing 1.1 require to demonstrate that theorems hold (1), each event preserves invariants (Induction (2)), each event can be observed (feasibility (3)) and variant decreasing (convergence (4)).

The proof obligations define what is to be proved for an Event-B model. They are automatically generated by a Rodin Platform tool called the proof obligations generator. This tool analyses contexts and machines. It decides then what is to be proved in these components with respect to the predefined list of proof-obligations (see Listings 1.1 and 1.2). The proof obligations generator

(5) Event Simulation (SIM)	$A \wedge I^A(x^A) \wedge J(x^A, x^C)$ $\wedge G^C(x^C, \alpha^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge \boldsymbol{BAP^C(x^C, \alpha^C, x^{C\prime}) \Rightarrow BAP^A(x_A, \alpha^A, x^{A\prime})}$
(6) Guard Strengthening (GS)	$A \wedge I^A(x^A) \wedge J(x^A, x^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge \boldsymbol{G^C(x^C, \alpha^C) \Rightarrow G_A(x^A, \alpha^A)}$
(7) Invariant preservation (INV)	$A \wedge I^A(x^A)$ $\wedge G^C(x^C, \alpha^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge BAP^C(x^C, \alpha^C, x^{C\prime})$ $\wedge \boldsymbol{J(x^A, x^C) \Rightarrow J(x^{A\prime}, x^{C\prime})}$

Table 1.2: Event-B refinement proof obligations

produces a number of logical sequents, which are transmitted to the provers for automatic or interactive proof.

Refinements preserve the relation between the refining model and the refined model. Abstract and concrete state variables are linked by introducing *gluing invariants*. Mathematically, the refinement PO formalise the condition of proving the simulation relation between abstract and concrete machines.

The PO generated for proving the correctness of a refinement are described in Listing 1.2. Two additional relevant PO need to be discharged. First, as refinement in Event-B is defined as a simulation relationship, the simulation PO (SIM) (5) is defined to assert that the refined (concrete) event evt^C simulates the corresponding abstract event. Second PO, guard strengthening (GS) (6), expresses that the refined events may strengthen the abstract guards G^A with another guard G^C to allow and specify more concrete conditions that shall be stronger than abstract ones. In addition, the invariant preservation PO (INV)(7) shall hold in the refined machine as well. More details on proof obligations can be found in [3].

1.3 Event-B Theories Extension

1.3.1 Extending Event-B Language with Theories

Event-B language may be qualified as *low-level modelling language* since it provides basically set theoretical concepts for modelling like sets, constants, membership predicate and operations like union and intersection among others. In addition, the modelling language provides the before-after predicate transformer (substitution operator). This language is as expressive as typed set theory providing a framework for modelling systems. However to handle more elaborate and high-level concepts, mathematical extensions may be defined to support Event-B basic notation and semantics. The extensions are Event-B theories providing primitives for defining and using advanced mathematical objects [7, 39].

Close to other formal languages module constructs like Coq modules [26], Isabelle/HOL theories [131] or PVS [134], this capability is suitable for high-level modelling as it provides more structured Event-B specification and open the possibilities for reusability and sharability.

Event-B theories are extensions allowing the definition of new data types and operators manipulating these data types. Moreover, they permit the definition of proof rules enhancing the automatic proving process.

An important feature of Event-B theories is **genericity**. Indeed, the theory mechanism allows us to define type-parametric modules reusable in several Event-B models. For example an Event-B generic theory of inductive lists may be defined such that it provides operator returning the length of a list and another operator for concatenating two lists. Such theory may be used by several Event-B models by instantiating differently the operators, for example a Event-B model may instantiate the theory with the integers $\mathbb{Z}$ and another model with booleans `BOOL`.

Additionally, the soundness [39] of theories is achieved through the definition of soundness proof obligations generated following the standard approach of Event-B and their proofs are discharged using Rodin provers. Only the soundness of the rewrite rules is handled; the soundness of the theory axioms shall be ensured externally [7, 39].

The Event-B standard library [2] includes `BasicTheory` project comprising theories of `BinaryTree`, `BoolOps`, `List`, `PEANO`, `SUMandPRODUCT` and `Seq`. Second `RelationOrderTheory` project provides theories of `Connectivity`, `FixPoint`, `Relation`, `Well_Fondation`, `complement` and `galois`. Also it defines the reals in `RealTheory` project. Additionally the library provides three simple Event-B models that use some of the theories: `Data`, `Queue` and `SimpleNetwork` projects.

1.3.2 Event-B Theories Structure

Listing 1.3 presents the skeleton of Event-B theory. It clearly recalls an algebraic specification involving data types and operators. The first clause of an Event-B theory is the `IMPORT` clause which provides a mechanism to reference other theories elements. Next, the second clause is `TYPE PARAMETERS` which is intended to contain a list of placeholders dedicated to contain concrete types at instantiation in Event-B contexts or machines.

Event-B theories have an algebraic structure where data types are defined and operators manipulating these data types are declared and specified.

- `DATATYPES` clause: data types are defined in `DATATYPES` clause. They are associated with `constructors`, which are operators to build inhabitants of the defined type. Event-B theories allow us to define inductive data types as list data structure or trees. The `DATATYPES` clause may equally used to define records and enumeration data structures.

[2] https://wiki.event-b.org/index.php/Theory_Plug-in

```
THEORY TheoryName
IMPORT Theory_1, ...
TYPE PARAMETERS T_1, T_2, ...
DATATYPES
  Datatype1(T_1, ...)
  CONSTRUCTORS
    cstr1(p: T_1, ...)
OPERATORS
operator1 < nature > (p_1 : T_1)
  well-definedness WD(p_1, ...)
  direct definition D
  ...
AXIOMATIC DEFINITIONS
  AxiomaticDefinitionsName_1
    Types AT_1
    Operators
      operator1 <nature> (p_1 : T_1)
        well-definedness WD(p_1, ...)
    Axioms Axm_1, ...
THEOREMS Thm1, ...
END
```

Listings 1.3: Event-B theories structure

- `OPERATORS`: the clause `OPERATORS` allows us to define a collection of two sorts of operators: expression and predicate operators. Expression operators must return an element of valid type and predicate operator are defined using first-order logic formulas. Event-B theories provide a second option to define operators: axiomatically defined operators.
- `AXIOMATIC DEFINITIONS` clause: this clause provides the ability to define abstract types and operators. Roughly speaking, `AXIOMATIC DEFINITIONS` may be compared to classical Event-B contexts where types, constants, functional and predicate operators may be defined in an axiomatic manner. Axiomatic definitions allows to define mathematically objects that are not easily constructible from set theory primitives, or when the internal structure of such object is irrelevant and only the abstract properties are important. A good example is the theory of real numbers, indeed an axiomatisation relying on the axiom of the least upper bound is provided in the standard library which is useful for specifications not requiring constructive definitions like Cauchy's sequences or Dedekind's cuts. [3]. Other axiomatic theories may be defined as well.
- `THEOREMS` clause: Theorems are important part of an Event-B theories, they assert true statements about operators or a special combination of

[3] https://wiki.event-b.org/index.php/Theory_Plug-in

operators. The theorems are generic if the domain of the quantified variables are the type parameters. The `THEOREMS` clause generates a number of PO to ensure that the formulas are well-defined (WD) and provable (S-THM) in the theory.

Another feature of Event-B theories is the ability to extend the proof capability of Event-B by specifying generic rewrite rules associated with some theory. This rules may be integrated in the prover tactics and run on PO that contain symbols of the theories like variables typed with the theory data types or expressions involving occurrences of the theory operators.

1.3.3 Well-Definedness

Event-B theories has an important notion: the **well-definedness** associated with operators. It is a generalisation of the well-definedness concept[4] attached to a number of classical Event-B operators like arithmetic division and the cardinality operator. For example, whenever a division is applied, Event-B requires according to mathematical basic rules that the divisor not to be equal to zero. A second example is the `card` operator, indeed this operator is defined exclusively for finite sets therefore whenever is applied to some set a PO is generated asking to prove that the single argument is finite.

According to [6], well-definedness describes the circumstances under which it is possible to introduce new term symbols by means of conditional definitions in a formal theory as if the definitions in question were unconditional, thus recovering completely the right to subsequently eliminate these symbols without bothering about the validity of such an elimination. It avoids describing ill-defined operators, formulas, axioms, theorems, and invariants. In Event-B, each formula is associated to generated well-definedness POs [107]. These POs ensure that the formula is well-defined and that two-valued logic can be used.

A WD predicate $WD(f)$ is associated with each formula f . This predicate is defined inductively on the structure of f. For example, if we consider a and b being two integers, P and Q two predicates, f of type $P(D \times R)$, the following WD definitions can be written as $WD(a \div b) \equiv WD(a) \wedge WD(b) \wedge b \neq 0$, $WD(P \wedge Q) \equiv WD(P) \wedge (P \Rightarrow WD(Q))$, $WD(P \cup Q) \equiv WD(P) \wedge (P \cup WD(Q))$ or $WD(f(a) \equiv WD(f) \wedge WD(a) \wedge a \in dom(f) \wedge f \in D \pfun R$ where $\pfun$ denotes a partial function. In the proofs of other POs, WDs are added as hypotheses [3].

The Event-B theories enable to define well-definedness conditions to user-defined operators. Event-B theories syntax offer a field to express a logical predicate associated as well-definedness condition to some operators. The well-definedness condition may involve the operators arguments by restricting their domain, also other operators may be used to build the well-definedness condition. The use of the operator in expressions generates `WD` PO. This PO must be discharged to ensure that the operator is used correctly; it means that the expression where the operator is used leads to defined results.

[4] `https://www3.hhu.de/stups/handbook/rodin/current/html/mathematical_notation_introduction.html`

Handling WD conditions and partial functions ($\nrightarrow$) definitions in proofs and proof systems is not new. The paper of C.B. Jones [95] clearly highlights the importance of dealing with such definitions. In formal proof systems, it has been addressed in different manners using two-valued and three-valued logic (with weak and strong equality), subset types, denotational approaches, type-correct conditions of total functions, etc. [23, 154, 107, 6, 131, 134]

This mechanism of associating well-definedness conditions to operators is essential to the contributions presented in Part II of this manuscript. It allows to represent dependent types that are not natively supported by Event-B.

1.3.4 Importing Theories

The original goal of Event-B theories is to allow for mathematical extensions of the Event-B language [7, 39]. As a consequence, libraries of structured formal specifications that can be reused across models are available as Event-B theories. This is possible thanks to the importing capability provided in the theory component. Indeed, theory hierarchies may be defined to reduce the complexity of a specification.

Event-B theories may be imported into models composed of contexts and machines. The models are said to instantiate the theory since they provide concrete types to the type parameters of the theory. Additionally, the models may import all the primitives of the theory: data types, operators, axioms. From the proof point of view, the proof rules and the rewrite rules can equally be applied when discharging some POs.

This manuscript includes the usage of a hierarchy of theories to define behavioural domain-specific analyses (see Chapter 7). It illustrates the interest and the benefits of building upon specified and proved theories.

1.4 IDE: Rodin Platform and Plug-ins

The thesis relied intensively on Rodin platform and the Theory Plug-in to carry out the modelling and proving experiments. However, there exist other plug-ins dedicated to various tasks [5].

Indeed, modelling in Event-B, investigating and verifying models is allowed by the Rodin Platform[4] which is an open source IDE based on Eclipse for dedicated to Event-B formal method. It offers resources for model edition, automatic PO generation, project management, refinement and proof, model checking, model animation and code generation. Several provers like SMT solvers are available in Rodin.

Designing Event-B theories is supported by Theory Plug-in which is available for Rodin platform. It provides an editor to create, modify Event-B theories and it relies on the Rodin theorem provers to prove the theorems and the proof obligations previously mentioned.

[5] https://wiki.event-b.org/index.php/Rodin_Plug-ins

Chapter 2

Domain Knowledge in Formal Modelling

This Chapter contains:

2.1 Modelling Domain Knowledge 22
 2.1.1 On the Importance of Domain Knowledge 22
 2.1.2 On the Lack of a Generic Approach 23
 2.1.3 Ontologies as Domain Knowledge Model 24
2.2 The Ontology Formalism 25
 2.2.1 Fundamental Characteristics 27
 2.2.2 Semantic Annotation Using Ontologies 28
 2.2.3 Ontologies for Engineering Contexts 28
2.3 Standards as Domain Knowledge 29
2.4 Synthesis and Conclusion 31

This chapter reviews the work involving modelling of domain knowledge in formal methods. Several articles addressed the issue of systematically modelling and referencing domain knowledge in formal models [15]. Handling domain knowledge system engineering activities, such as modelling and verification phases, is essential to ensure reliable and secure systems. This chapter is divided into four sections. Section 2.1 is dedicated to review the work on formal methods addressing the issue of taking into account the domain knowledge associated with a system. Section 2.2 is dedicated to the study of the work achieved on ontology-based modelling and its role in formalising engineering domains. Section 2.3 enumerates a number of approaches that addressed the challenge of formally checking conformance since it is a direct application of the explicit modelling of domain knowledge as discussed in Chapter 8. Last, Section 2.4 concludes this section by presenting a list of features required for a framework to allow explicit modelling of domain knowledge in formal specification.

2.1 Modelling Domain Knowledge

2.1.1 On the Importance of Domain Knowledge

To motivate this section, let us consider this simple example that illustrates the risks of missing information implicitly assumed when modelling. Let us consider, two state variables: float variable measuring the altitude of an aircraft in miles whilst another one denoting the value of the speed of the aircraft displayed on the cockpit in knots. When the knowledge about units is omitted or it is hard encoded in the designed formal model, then it *is not explicitly described*; it remains tacit in the mind of the model designer. This situation suffers from several downsides: (1) the operations on the variables may produce meaningless results. For example, adding these two floats (altitude plus speed) is nonsense. (2) the second disadvantage is the inability of reuse and sharing of the specification. Therefore, the definition of the arithmetic operators on these variables is necessary every time since they are not available as reusable primitives.

The issue of systematically separating domain knowledge and contextual constraints from system-specific requirements has been an essential challenge in system engineering in general and software engineering in particular. Indeed, taking into account the domain knowledge requirements and assumptions in the software engineering process has been considered an important challenge in the area of software systems modelling and analysis. The triptych described in [90], [171], [31] identifies three main parts of the software development process: domain description, requirements prescription and software design. $\mathbf{D}, \mathbf{S} \vdash \mathbf{R}$ expresses a formal deduction, where $\mathbf{D}$ represents the domain concepts in form of properties, axioms, relations, functions and theories; $\mathbf{S}$ represents a system model; and $\mathbf{R}$ represents the intended system requirements. This entailment states that the given domain description ($\mathbf{D}$) and the system model ($\mathbf{S}$) yields logically the given requirements ($\mathbf{R}$). In similar vein, the article introduces this notations [90] $\mathbf{E}, \mathbf{S} \vdash \mathbf{R}$ where $\mathbf{E}$ is the given environment, $\mathbf{S}$ is the specification that is description of designed system; and $\mathbf{R}$ represents the requirements the system should enjoy. The proposed structure must respect the distinction between system and the physical environment, and the environment properties must be achieved by the modelled system [90].

The first common knowledge to be explicitly formalised is mathematics. Indeed, several research projects and approaches [27, 28, 29] aimed at formalising mathematical theories that are applicable in the formal developments of systems. These theories are helpful for building complex formalisations, expressing and reusing proof of properties. The need for handling domain knowledge other than mathematics has been identified so far by the software engineering community [91, 171, 170] following the triptych paradigm promoted by D. Bjørner and A. Eir [33]. Moreover, a number of works [14, 30] advocated that it is highly desirable to define the domain knowledge associated with a system in an *explicit* way to improve the quality of the development process and to accommodate the new changes in the system requirements by restructuring the formal model.

In [120], the authors highlighted the need for separating and integrating

explicit semantics of application domain (domain knowledge) into the formal development process. Traditionally, in Event-B developments, domain descriptions are implicit and usually shared half way between the requirements model and the system model. In the work [100, 101], contextual knowledge and environment constraints integration in formal models is fostered and illustrated. Indeed, they proposed a methodology for integrating contextual knowledge in Event-B formal modelling process where domain knowledge is classified into constraints, hypotheses, and dependencies.

Observation (1) Explicit formal modelling of domain knowledge is an important challenge in the formal methods research and practice. Additionally, the textbook [85] identifies the issue of formalising domain knowledge and integrating this domain knowledge into design models. More recently this objective is advocated in [14] where the authors argues the advantages of handling domain knowledge explicitly in formal design models. In conclusion, there is a need *to define a framework for explicit modelling of domain knowledge.*

2.1.2 On the Lack of a Generic Approach

The book [15] compiled several works and reflections of 30 researchers issued from both academia and industry from America, Asia, Australia and Europe related to the challenge of making explicit domain knowledge in the formal developments related to several application domains. A methodology for a refinement-based development of control systems facilitating the identification of security requirements that should be fulfilled to satisfy safety goals has been proposed and illustrated in [162]. The author adopted the four-variable model defined in [136] and defined a approach with three refinements. The first step is to set the abstract specification describing the overall behaviour of the system; second this specification is translated to a general Event-B machine. Last the three refinements are defined for introducing data, specifying controller logic and attack modelling. Moreover, explicit modelling of domain knowledge was addressed in the medical devices engineering area; an annotation mechanism has been proposed for explicit modelling of domain-specific knowledge integration in formal system modelling

Observation (2) Despite of the abundance of approaches to address the challenge of explicit modelling of domain knowledge and the multiplicity of cases studies for various experiments, the approaches aforementioned lacked systematic and generic method for formalising and transferring domain knowledge to design models. Indeed, all the approaches aforementioned don't provide a language for describing the domain knowledge instead they provide methodological guidelines to define domain knowledge using ontologies and axioms, and where theorems are used to validate domain-specific properties.

2.1.3 Ontologies as Domain Knowledge Model

Different projects have aimed at providing principles, techniques and tools to streamline the integration of domain knowledge in formal modelling. For example, the French ANR IMPEX research project [1] set the objective of making explicit domain knowledge in design models. Many research and technical articles have been produced [2]. The project focused on ontologies which are formalised as theories with sets, axioms, theorems and reasoning rules. They are integrated to design models through an annotation mechanism [9].

The Event-B method community has been noticeably active in addressing the issue of explicit modelling of domain knowledge. Indeed, a great effort has been invested into systematically exploring methodologies for handling domain knowledge in formal modelling. In [7, 39], Event-B is extended by introducing Event-B theories in the form of the *Theory Plug-in*. This extension enables the specification of domain-specific theories, such as theories for hybrid systems developed in [60, 59, 58]. Indeed, the article [111] advocates the adequacy of Event-B for modelling the domains except in some areas, temporal properties mainly. Yet, the authors defined, in the article [86], a set of proof rules to reason about important classes of liveness properties. Moreover, mathematical theories (groups, reals, differential equations) for the Event-B method were developed and used in formal system developments in [59, 58]. Additionally, a methodology with the supporting tool OntoML was proposed [123, 16] for generating Event-B models from OWL-described ontologies. Furthermore, [76, 77] proposed an ontology description language that was used in the development of case studies. Event-B contexts were used to describe the ontology structure, and the context extension operation allows the definition of a particular ontology. Important properties that must be proved are manually annotated as theorems. Theorems are also used for formalising the compliance of the design model to the domain ontology. The articles [124] presented a plug-in integrated into the Rodin platform implementing two approaches to formalise ontologies described by ontology description languages (OWL, PLIB, ...) using set theory and first-order logic supported by Event-B. The interest of this formalisation is to enrich the specification and verification process using the Event-B method, by integrating data and knowledge models described in ontologies.

Besides, many formal languages addressed the challenge of making explicit domain knowledge methodologies and approaches for dealing with domain knowledge. Indeed, formal techniques like Coq [26], Isabelle/HOL [131], PVS [134], Event-B [3] CASL [125] and RAISE [29, 31, 32] witnessed several works dealing with this issue. Other modelling frameworks, such as DOL, CASL [125] and RAISE [29, 31, 32], addressed the railway systems, shipping and logistics domains to describe domain knowledge. The issue of addressing complex structured specifications by combining and extending simpler ones is addressed in [146]. Indeed, the authors advocated that an understanding of a large spec-

[1] https://anr.fr/Project-ANR-13-INSE-0001

[2] https://hal-anr.archives-ouvertes.fr/search/index/?q=*&anrProjectReference_s=ANR-13-INSE-0001

ification might be achieved via an understanding of its components, and the components of large specifications may be reused. Moreover, the structure of a specification conveys intangible but important aspects of the conceptual structure of the problem domain, such as the degree to which entities and concepts described in the specification are interrelated. In [50], the authors discussed a two-layered language for expressing and specifying contexts that are based on higher-order logic of the Coq formal system as a lower layer and ontology language as an upper layer. Indeed, the higher-order KDTL language [21, 49] supports the definition of new contextual categories and facts based on low-order context. The language provides support for the comparability of diverse and non-countable information, as well as numeric data.

Engineering domain ontologies as defined in [138, 94, 11] proved to be efficient in capturing engineering domain features and to model the associated domain knowledge. Indeed, engineering domains require other modelling capabilities like arithmetic operations or context-dependent properties and associated proof rules (see [94, 14] for more details). To handle engineering knowledge, first-order logic with arithmetic may be used as ontology modelling language. However, this richer expressive power leads to semi-automatic proofs requiring interactive proof effort[3].

Observation (3) A generic ontology modelling language for formalising domain knowledge would provide flexibility and interoperability for the exchange of domain knowledge. However the ontology modelling language must provide primitives for expressing a special feature of domain knowledge related to engineering contexts like arithmetic.

2.2 The Ontology Formalism

The term *Ontology* comes from the discipline of philosophy that is concerned with the study of being or existence. In philosophy, one can talk about an ontology as a theory of the nature of existence. In computer and information science, ontology denotes an artefact that is designed for a purpose, which is to enable the modelling of knowledge about some domain, real or imagined [73].

> *In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application ...* [4]

[3] *Automatic* reasoners (decidable logics) like Pellet [153] or Racer [75] apply to *less rich* OML than the one offered by Event-B theories

[4] https://tomgruber.org/writing/definition-of-ontology

Different domain knowledge description and modelling paradigms which are at the core of the knowledge representation area were proposed. Indeed, representing knowledge as ontologies has been extensively studied in the literature and other application areas, such as semantic web, artificial intelligence, information systems, system engineering and so on exist. Therefore, several modelling languages are available like DAML+OIL[48], RDF[37], OWL[163] and ISO 13584[139]. Moreover, many tools are available for creating repositories like JENA-SDB [168] SW-Store [2], trpileSotre [80], OntoDB [54, 11] or OntoHub [47]. Furthermore, tools like Protégé [5] [102] or PlibEditor [6] provide browsing capabilities in knowledge management and also query languages like RQL [98], SPARQL [140], OntoQL [93, 11]. In addition, reasoners like Pellet [153], RACER [75] or KAON [126], annotators like CREAM [79], Terminae [56] or SAWSDL [103] and translators [38, 157] have been proposed. Domain ontologies have been described for domains such as encyclopedia [87], logistics, rivers, canals [29], transportation systems [29, 172], geology [12], electronic components [88] and bio-informatics [22] etc.

In the context of database systems, an ontology can be viewed as a level of abstraction of data models, analogous to hierarchical and relational models, but intended for modelling knowledge about individuals, their attributes, and their relationships with other individuals[127]. In fact, ontologies are typically specified in languages that allow abstraction away from data structures and implementation strategies; in practice, the languages of ontologies are closer in expressive power to first-order logic than languages used to model databases. For this reason, ontologies are said to be at the "semantic" level, whereas database schema are models of data at the *logical* or *physical* level. Due to their independence from lower-level data models, ontologies are used for integrating heterogeneous databases, enabling interoperability among disparate systems, and specifying interfaces to independent, knowledge-based services. In the technology stack of the Semantic Web standards, ontologies are called as an explicit layer. There are now standard languages and a variety of commercial and open-source tools for creating and working with ontologies.

In the context of requirement engineering, the new requirements elicitation method ORE (Ontology-based Requirements Elicitation) was proposed in the article [96], where a domain ontology can be used as domain knowledge. The authors advised a domain ontology to play a role in the semantic domain which gives meaning to requirement statements by using a semantic function. Indeed, by using inference rules on the ontology and quality metrics on the semantic function, an analyst can be decide which requirements should be added for improving the completeness of the current version of the requirements and/or which requirements should be deleted from the current version for keeping consistency. Additionally, the authors [97] proposed a method and a tool to enhance an ontology of domain knowledge for requirements elicitation by using Web mining.

[5] http://protege.stanford.edu/
[6] https://www.iso.org/standard/43423.html

Ontologies enable an explicit representation of a domain of application of the system under design. Explicit means fully revealed or expressed without ambiguity, whilst implicit means implied or expressed indirectly or tacit [14]. However, the meaning of these two words may be used in an inconsistent way, within the computer science and software engineering communities. For example, in logic and belief models [108] a sentence is explicitly believed when it is actively held to be true by an agent, and it is implicitly believed when it follows from what is believed. In contrast, semantic web [159] or system engineering assign a slightly different meaning, i. e. semantics can be implicit, existing only in the minds of the humans [...]. They can also be explicit and informal, or they can be formal. For example, the Explicit Semantic Analysis (ESA) [67] interprets semantics of unrestricted natural language texts and represents meaning in a high-dimensional space of concepts derived from Wikipedia, the largest encyclopedia in existence. The meaning of any text is explicitly represented in terms of Wikipedia-based concepts. The requirements engineering community uses the terms to distinguish between declarative (descriptive) and operational (prescriptive) requirements [161], where they acknowledge the need for a formal method for generating explicit, declarative, type-level requirements from operational, instance-level scenarios in which such requirements are implicit.

The last definition from the requirement engineering community will be used as a *reference definition* of explicit and implicit terms throughout this manuscript. Indeed, the Event-B theories feature declarative type-level aspects thanks to generic algebraic structure (data types, operators, axioms and theorems) and the Event-B models feature operational and instance-level aspects thanks to its operational semantics (trace-based semantics).

2.2.1 Fundamental Characteristics

Ontology definitions meet three fundamental criteria [94]:

Formality. An ontology is a conceptualisation expressed in a modelling language. It has an underlying formal semantics and it supports reasoning. As for modelling languages, the semantics of ontology modelling languages is expressed using satisfaction and entailment relations.

Automatic or semi-automatic reasoning techniques are associated with an ontology modelling language. They allow the verification of instances through the syntactical entailment ($\models_O$) and reasoning thanks to the semantic entailment ($\vdash_O$). Consequently, the verification of properties expressed on concepts and individuals defined by the ontology becomes possible, thanks to the use of automatic reasoning techniques.

Consensuality. An agreement on the conceptualisation defined by an ontology must be reached for a large community of users. This community is not limited to the users or developers of a specific application: it includes all potential users and developers of other applications related to the conceptualised

domain. Therefore, an ontology will be shared by several applications and design patterns. For example, product ontologies compliant with ISO 13584 (PLIB) [139] are defined through a formal standardisation process. They are published as ISO and/or IEC international standards. This criterion excludes conceptual models defined for a specific application.

Ability to be referenced. Each concept defined in an ontology is associated with an identifier provided to allow applications to reference that concept from any environment. Moreover, this concept can be referenced regardless of the ontology model implemented to describe that concept. In this manuscript, the reference ability will correspond to the ability of formal models to reference concepts of ontologies.

2.2.2 Semantic Annotation Using Ontologies

Labelling in the semantic web offers two advantages over these systems: *improved information retrieval* and *improved interoperability* [158]. Consider a set of entities that exist in a given corpus. These entities can be words or sentences in a document, images or videos, entities in a design model, etc. By annotation we mean the connection that can exist between a concept of the ontology (class, instance, property, etc.) and an entity of the corpus under consideration. The annotation process consists in defining and applying a set of rules that lead to the creation of annotations. This process can be fully automated, semi-automated with user validation, or fully interactive. Automatic annotation has proven to be powerful in the semantic web and natural language processing domain, since the entities of the corpus are words that appear in texts. Several tools (or annotators) have been developed for different ontologies and natural languages[79, 84, 17, 36]. Other approaches aiming at annotating images and multimedia documents have also been developed [45]. In the area of system design, the goal of model annotation is to increase the interoperability of models. Consensus domain ontologies are used by different system models that correspond to different technical views. The annotations enable the designer to link different entities from different system models to the concepts of the ontology. The inferences at the ontology level allow to verify some domain properties. Model annotations are created using semi-automatic and/or interactive approaches. Automatic annotation is not recommended in these application domains. For example, model annotations are created for Product Life cycle Management (PLM) models in [109], for petroleum engineering models in [112, 25], or for aerospace systems modelling in [79]. All of these examples use controlled annotation techniques, either semi-automatic or interactive.

2.2.3 Ontologies for Engineering Contexts

Ontologies addressed in this thesis pertain to the engineering realm. The PLIB [139, 92, 137] ontology model advocates the use of strong typing with a rich type

system, property derivation with algebraic operators corresponding to the defined types, first-order logic and set theory as a constraint language, CWA and context-dependent properties. Like in usual engineering practices and unlike OWL, additional models may be added to a technical object description. Indeed, a set of different functional models, each one representing a particular view or discipline-specific representation (e.g., safety, real-time, energy consumption, geometry procurement, simulation, etc.) can be associated with a given technical object described within the PLIB ontology model.

Finally, several domain ontologies based on this model already exist. Examples are ISO 13584 and ISO 15926 (e.g. mechanical fasteners, measure instruments, cutting tools) and IEC 61360 (e.g. electronic components, process instruments) series of ontologies developed within international standardization organizations (e.g. ISO, IEC) or national ones (e.g. JEMIMA [7] CNIS [8]) that cover progressively all the technical domains.

Benefits of Ontologies Benefits of adopting ontology formalism is enumerated as follows [133]:

- sharing common understanding among people or software agents
- enabling reuse of domain knowledge
- making explicit domain assumptions
- separating the domain knowledge from the operational knowledge
- analysing domain knowledge

2.3 Standards as Domain Knowledge

According to ISO, a standard is defined as *Standards are documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines or definitions of characteristics, to ensure that materials, products, processes and services are fit for their purpose* [89].

The use of standards has several potential advantages. It plays an important role in the development of any complex system, which includes both product-based and process-based development. The compliance checking of product with respect to standards documents is both time-consuming and difficult. Some work focuses on integrating standards into process development. In [61], the authors gave a brief account of the notations and method they have developed to support the use of the model and to describe a support environment. The contributions of the article are the identification of the issue of standards compliance; the development of a model of standards and support for compliance management; the development of a formal model of product state with associated

[7] Japan Electric Measuring Instruments Manufacturers Association,
[8] Chinese Institute for Standardization

notation; a policy scheme that triggers checks; and a compliance management view. The article [24] introduced a framework that, using Natural Language Semantics techniques, helps to process legal documents and standards to build a knowledge base to store their logic representations, and the correlations between them. The knowledge base allows legal experts assess what requirements of the law are met by the standard and, therefore, recognize what requirements still need to be implemented to fill the remaining gaps. An application of the framework is exemplified by comparing a provision of the European General Data Protection Regulation against the ISO/IEC 27001:2013 standard. In [43], the authors gave a general overview of the compliance checking field. First, they highlighted the benefits that compliance checking can bring to several digital transformation initiatives. Second, they defined the compliance control framework, which includes process modelling and execution. Finally, some areas in which compliance checking is relevant are highlighted. Similarly, [110] showed how to implement the compliance relationship on transition systems. The computability of this relation relies on the composition of two operators : the reduction relation, and the fusion function of the acceptance graphs associated with the compared transition systems. It is formally demonstrated and illustrated by a case study. Nair et al. [128] presented the results of a questionnaire survey examining the state of practice in safety evidence management. The results are based on 52 valid responses from 11 different sectors and 15 countries. The survey explored industry perspectives and practices on (1) the types of safety data used, (2) the processes and means for managing data changes, (3) the data structuring and assessment techniques used, and (4) the challenges faced by practitioners. The paper showed that V&V artefacts such as the V&V plan, test results, and test case specifications are among the most commonly used as security evidence, demonstrating the importance of V&V for security evidence. However, some verification techniques such as model checking and theorem proving are used in low numbers in the industry.

In recent years, assurance cases have been used in critical domains to establish system safety by presenting appropriate arguments and evidences [99, 145]. In [70], eliminative induction-based framework was presented, the principle (proposed by Francis Bacon) that confidence in the truth of a hypothesis (or claim) increases as reasons for doubting its truth are identified and eliminated. Possible reasons for doubting the truth of a proposition (defeaters) arise from the analysis of a reliability case using the concepts of doubt-inducing reasoning. Finally, Bacon's concept of probability provides a measure of confidence based on how many defeaters are identified and eliminated. Furthermore, the article [74] explored the main current approaches, and proposes a new model for quantitative confidence estimation based on Belief Theory for its definition, and on Bayesian Belief Networks for its propagation in safety case networks. Additionally, the article [72] proposed a formal argument evaluation of the the concept of an assurance case argument originally introduced by Toulmin[156]. The approach provide a mean of measuring how much justifiable conclusions are with respect to the arguments. Wassyng et al. [165, 164] proposed a product domain assurance case template as a standard for the development and licensing of med-

ical devices within that product domain. The authors developed an assurance case template, where sub-claims and details of generated evidence need to be plugged in appropriate placeholders. The article [173] presented an Event-B formalisation and verification for the ARINC 653 standard, which provided a standardised interface between safety-critical real-time operating systems and application software, as well as a set of functionalities aimed to improve the safety and certification process of such safety-critical systems.

Intuition (4) Standard conformance may be addressed and formalised as explicit domain knowledge. Indeed, the standard is regarded and modelled as domain knowledge which is referenced from the design models. Therefore, the properties and requirements described in the standard documents may be transferred to the model hence it allows to ensure its conformance.

2.4 Synthesis and Conclusion

This chapter reviewed works on domain knowledge in formal modelling, and established that ontology formalism is a good candidate for describing domain knowledge. Last, several works on conformance checking have been studied.

This survey reveals that integrating domain knowledge requirements in the early stages of the system development is of utter importance (**observation (1)**) to ensure high-quality modelling and verification. On the one hand, such knowledge is provided implicitly during the system development by making some assumptions about an environment and some past experiences. Commonly, such implicit domain knowledge often shows some contradictory results, which may lead to a system failure state. On the other hand, formal methods do not provide primitives for describing explicitly domain knowledge (**observation (2)**). Another observation is that several research projects and approaches aimed at formalising mathematical theories applicable to the formal development of systems. These theories help building complex formalisation and expressing and reusing proof of properties. Usually, these theories are defined within contexts, imported and/or instantiated. They usually represent the implicit semantics of the systems, by types, logics, algebras, and so on. Additionally, **observation (3)** emphasises the adequacy of ontology formalisms to express and describe domain knowledge requirements.

From the **observations (1), (2) and (3)**, it is clear that there is a need for a generic framework and systematic approach addressing the formal and explicit description of domain knowledge and allowing the requirements to be transferred to the models. Likewise, the framework shall provide these features:

- **Feature-1:** Primitives for modelling domain knowledge concepts and associations, preferably generic ones.

- **Feature-2:** Primitives for expressing constraints of a domain of knowledge,

- **Feature-3:** A mechanism for referencing domain knowledge primitives from design models,
- **Feature-4:** A mechanism for transferring safety properties (static properties) and behavioural properties (dynamic properties) required by the domain knowledge to the formal models.

Intuition (4) asserting that standard specification may be viewed as domain knowledge is the basis for the contribution concerning the methodology of checking conformance to standard specification (see chapter 8).

Chapter 3

Interactive Critical Systems

This Chapter contains:

3.1 Formal Methods for Interactive Systems 34
  3.1.1 Interactive Systems Characteristics 34
  3.1.2 Formal Design of Interactive Critical Systems 36
3.2 Interactive Systems Development 39
3.3 The Context of The FORMEDICIS project 42
3.4 Synthesis and Conclusion 42

This chapter reviews the work related to formal methods tailored to the design and verification of interactive critical systems. In this thesis, interactive critical systems engineering is used as a pool of case studies for experimenting and exemplifying the methodologies and the frameworks of explicitly modelling of domain knowledge in formal modelling which is our central focus. Therefore, the methodology for modelling domain knowledge requirements in the process of system formal modelling may be used in other areas such as railway systems or autonomous vehicles.

In this chapter, section 3.1 reviews techniques and tools dedicated to formal modelling of interactive critical systems. Then, section 3.2 summarises two important architectures established in interactive critical systems design, which are relevant to the work of domain knowledge explicit representation. Next, section 3.3 discuses the FORMEDICIS project context which was the context of this PhD thesis. Moreover, FORMEDICIS project was dedicated to designing and verifying interactive critical systems. Finally, section 3.4 concludes the chapter.

3.1 Formal Methods for Interactive Systems

3.1.1 Interactive Systems Characteristics

Critical Interactive Systems Design. Development processes of critical interactive systems in disciplines such as aeronautics, space and transports are inspired by traditional software development processes. In addition, international, widely adopted standards that take into account the safety and security requirements of the systems under construction are required by certification authorities. In particular, DO178C standard [66], in aeronautics, defines very strict rules and instructions that must be followed to produce airborne software products, embedded systems and their equipment. For instance, they need to show that high-level requirements are in conformance with low-level descriptions of software behaviour. Unfortunately, they do not explicitly mention how users' needs are identified and represented, making them partly not adequate for interactive systems development. For this reason as well as inherently different nature of interactive systems[147], other documents deal with the part of the interactive system (such as [8], in aeronautics and [62] in space domains). In aeronautics, some standards are dedicated to the description of interactive applications. For instance, [18] makes explicit the properties and abstract behaviour of all the interactive components deployed in interactive cockpits. Beyond, it also defines the communication protocol between the user interface server and the interactive applications that are exploited by the flying crew [20]. Note that some tools recently appeared to enhance the specification, coding and certification stages of these interactive systems offering WIMP interfaces [160]. But these tools, for instance, Scade Display[1], deal mainly with information displays and do not represent explicitly the states, events and behavioural evolution of the components of the interactive systems (servers, widgets and user applications) as argued in [20]. The paper proposes to merge the elements of these three types of approaches: development processes highlighting the importance of requirements and formal description techniques (e.g. [142]), user- and usage-related requirements and needs (e.g. [66]) as well as WIMP user interfaces standards (e.g. [18]). Indeed, all these elements contribute to the production of usable, dependable and certifiable interactive systems [129]

Furthermore, classical software development methodologies recommend beginning with requirement elicitation which becomes the basis for the production of software artefacts. Waterfall [144] or V cycle [113] methodologies are typical examples of these safety-critical system development methodologies. These processes were recommending iteration-based development processes, as advocated by Winston W. Royce's 1987 article[144]:

> *If the computer program in question is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is the second version insofar as critical design/operations areas are concerned.*

[1] https://www.ansys.com/products/embedded-software/ansys-scade-display

Indeed, interactive software development processes adoption dates as far as back the mid-1950s. Prominent software-engineering thought leaders from each succeeding decade supported interactive and incremental practices, and many large projects used them successfully [105]. The seminal work [132] advocated the essential importance of usability goals, user characteristics, environment, tasks and workflow of a product, service or process are given extensive attention at each stage of the design process; it stated central interactive system issues, questions, and complex trade-offs facing designers, creators, and users of interactive systems. Moreover, the article [142] presented the notation of optimisation cycles and discussed techniques for user-participation integrated into a general concept of participatory software development optimisation of cycles in software development. (End-)user requirements, global and detailed task analysis and *formal specification* are identified as intermediate products.

Human in The Loop Human factors and ergonomics [166] are key element in developing interactive critical systems. Within the NASA, the following definition of human factors is used[2].

> *Human factors is an umbrella term for several areas of research that include human performance, technology design, and human-computer interaction. The study of human factors in the Human Factors Research and Technology Division at NASA Ames Research Center focuses on the need for safe, efficient and cost-effective operations, maintenance and training, both in flight and on the ground.*

Indeed, human factors are widely investigated [34, 167] while designing such systems, where important topics are established as training for advanced automation, cockpit errors and error reduction, management of cockpit workload, and general attitudes toward cockpit automation. Different methodologies are proposed for addressing both the criticality aspects and the user aspects. The formal syntax and semantics of the EOFM (Enhanced Operator Function Model) and an automated process for translating an instantiated EOFM into the model checking language Symbolic Analysis Laboratory. In [35], EOFM, an Extensible Markup Language-based, platform- and analysis-independent language for describing task analytic models, is introduced. An automobile cruise control example to illustrate how an instantiated EOFM can be integrated into a larger system model that includes environmental features and the human operator's mission. The system model is verified using model checking in order to analyse a potentially hazardous situation related to the human–automation interaction. In [55], the authors presented an approach to detect and cure conflicts in aircrafts pilots activities. The approach relies on three steps, the first step is to track the pilot's activities to reconstruct her/his behaviour using parameters and reference models of the mission and the procedures. The second step is to detect conflict in the pilot activity and this is linked to what is involved in the

[2] https://www.hfes.org/About-HFES/What-is-Human-Factors-and-Ergonomics

achievement of the mission. The third step is to design accurate countermeasures. Experimental results obtained from private and professional pilot have been conducted to validate the approach.

Opportunities & Challenges. Opportunities and challenges have been identified by the use of formal verification in the analysis of critical interactive computing systems in the article [42]. Three main challenges are discussed: (1) the accessibility of the modelling stage; (2) support for expressing relevant properties; (3) the need to provide analysis results that are comprehensible to a broad range of expertise including software, safety and human factors. A deep reflection on tools used and the problem hindering their accessibility was presented in the article [82]. It commented on tool developments that could lead to wider use of these techniques. It explored the role that existing methods and tools can play in analysing interactive systems and concrete examples involving the use of the PVS theorem proving assistant and the IVY toolset are given. The focus was put on the formulation and validation of models of interactive systems; the expression of user-related requirements, especially in the context of usability engineering and safety analysis; the generation of proofs that requirements hold and make sense when proof fails. Examples include standalone medical devices including examples from part of a safety analysis of a device leading to the product.

The state of affairs of formal methods dedicated to interactive critical systems fostered several lines of works by different teams across the world. Indeed, a number of tooled approaches using formal methods to address issues of designing and verifying interactive critical systems emerged and are still in development are reviewed hereafter.

Observation (1) Modelling the environment of the system under design is of utter importance to ensure its usability and high-quality. This requirement may be formalised as common domain knowledge and then transferred when constructing the system model.

3.1.2 Formal Design of Interactive Critical Systems

Significant work has been carried out to create tools that combine user-centred design and formal verification technologies for the modelling and analysis of interactive systems [81, 135, 57], and it is now agreed that it is the only way to integrate interactive systems in critical systems [71]. For examples, formal methods have been used to check functional requirements and safety requirements by developing models for interactive systems [83].

Event-B Formal Method. Event-B and ProB have been successfully applied to modelling interactive systems and validation of user tasks. In [10], the authors proposed a tooled approach based on Atelier B tool and the SUIDT model-based tool to address the cooperation between formal and experimental

HCI properties validation and verification. In [44], an approach to address the problem of user interface evolution because of the introduction of new interaction devices and/or new interaction modes is proposed. The authors considered that interface behaviours are modelled as labelled transition systems, a comparison between user interfaces is achieved by an extended definition of the bi-simulation relationship to compare user interface behaviours when interaction modes are replaced by others. Moreover, [69, 150] defined a methodology based on Event-B for developing an interactive system using a correct-by-construction approach. It supports a development of the model-view-controller (MVC) architecture. The whole approach has been illustrated on an industrial case study that illustrates the effectiveness of our proposed approach for developing an HMI. The EB2All [121, 149, 148] tool is used for code generation when the suitable refinement level is reached. In the previous works, scalability issues are addressed by proof-based techniques where refinement plays an important role in handling the complexity by developing models incrementally [5].

CIRCUS. A tooled suite for modelling and validation of interactive critical systems is provided by CIRCUS aka Computer-aided-design of Interactive, Resilient, Critical and Usable Systems [3]. It supports the development of two types of models. First, system models allows us to cover functional core, interaction and dialogue technologies. Second, hierarchical task models allowing us to describe user behaviour, user knowledge, strategies, information and equipment required to reach goals. The CIRCUS environment is intended to support multidisciplinary teams of software engineers, system designers and human factors experts and it enables the editing and simulation of task models. The tool can be exploited to ensure consistency, coherence and conformity between an envisaged or prescribed user task and the sequence of actions required to operate an interactive system. The notation used in the tool allows user goals and sub-goals to be structured into a hierarchical task tree. Mathematical operators can also be used, such as for modelling temporal relationships between tasks and special task types, for explicit representation of data and knowledge, for describing devices, and for representing errors, genotypes and phenotypes of co-operative tasks, etc.

PVSio-web. PVSio-web is an open source toolkit for model-based development of user interfaces. The purpose of the toolkit is to support multidisciplinary teams of user interface engineers, domain experts and software analysts. This support is achieved by integrating special components designed for different target users. (1) The Prototype Builder allows developers to create the visual aspects and logic of operation of a prototyped device. The visual aspect of the prototype corresponds to an interactive image of the device realised by web technology. The operating logic is developed in the language of the Prototype Verification System (PVS). (2) A simulator renders the appearance of

[3]https://www.irit.fr/recherches/ICS/documentation/

the prototype in a web browser. The behavioural logic of the prototype is executed in PVSio, a native component of the PVS system for animating PVS models. User actions on input widgets (e.g. button presses) are converted by the simulator into PVS representations that can be evaluated in PVSio then the results are displayed in a web browser using the prototype's output widget. This ensures that the appearance of the prototype is close to the appearance of the real system in the corresponding state. (3) Storyboard Editor facilitates the development of mock-up prototypes based on storyboards (storyboards). It can load various screen mock-ups, defines input widgets on them and associates the screen transitions with the user actions associated with the input widgets. It provides (1) a way to validate a model and to show that it faithfully represents the device, (2) a way to formalise requirements given in natural language and to demonstrate the benefits of the formalisation process, and (3) a way to prove the requirements of the model using readily available formal validation tools.

IVY. It is a tool for model-based analysis of interactive systems designs[4]. The tool consists of a set of plug-ins that serve as a front end to the NuSMV model checker [46]. The toolkit supports a notation, Modal Action Logic (MAL), that enables the specification of interactive systems and provides a set of property templates designed to aid the development of appropriate properties for analysing the model. The results, which include traces provided by the model checking analysis when a property fails to be satisfied, are depicted. The goal of IVY is to produce user-friendly representations and analysis tools for user interface developers. Moreover, the results could be communicated effectively within an interdisciplinary team of formal methods experts and software engineers. The IVY toolkit architecture is organised into an extensible set of interoperable components: MAL editor, property editor, trace visualiser and trace simulator.

Data Flow Languages. A synchronous data flow language, Lustre [78], is used for describing and analysing interaction mechanisms of user interfaces. In [52], a set of possible interactions is derived from an informal description of user interfaces, and the derived interactions requirements are further used for developing a formal model of user interfaces to analyse system interactions [53], and then for generating the test cases [51]. Furthermore, LIDL Interaction Description Language (LIDL) [106] is proposed for describing a formal description of user interfaces. In LIDL, the static description of user interfaces is defined by interfaces while the dynamic description of user interfaces is represented through interactions. The semantics of this language is also based on synchronous data flows similar to Lustre that makes the process easy for formal verification and code generation. Ge *et al.* [68] presented a formal development process for designing interactive applications for safety critical systems. In this development process, the LIDL language is used for describing user interfaces and S3 solver for analysing the described model.

[4] http://ivy.di.uminho.pt/

F3FLUID. The Formal Framework For FLUID (F3FLUID) was intended for the development of interactive safety-critical systems. In [119], the authors gave a detailed presentation of F3FLUID. This framework is based on FLUID aka Formal Language of User Interface Design which is a pivot modelling language defined in the FORMEDICIS project [5] enabling high-level system requirements for interactive systems to be specified in the FLUID language. This modelling language is specifically designed for handling concepts of interactive safety-critical systems, including domain knowledge through annotations called tags. Furthermore, FLUID defined an extension mechanism relying on the refinement of Event-B which allows an incremental design of interactive systems. Formal verification, validation and animation of the designed models are supported through different transformations of FLUID models into target formal verification techniques: Event-B for formal verification, ProB model checker for animation and Interactive Cooperative Objects for user validation. The Event-B models are generated from FLUID, while ICO and ProB models are produced from Event-B. The TCAS (Traffic Alert and Collision Avoidance System) case study was used as an example of the framework. Additionally, Multi-Purpose Interactive Applications (MPIA), was developed to illustrate the effectiveness of F3FLUID framework for the development of interactive safety-critical systems [151]. The ICO/PetShop framework [130, 19, 65] is used for animation of the presentation. Indeed, Petri nets-based verification procedures can connect interactive objects of the user interface to ICO models through Petri nets. By using the visual animation of the user interface, it is possible to run the modelled interactive application. These formal techniques and dedicated tools play a complementary role in the modelling process, and they provide support and feedback contributing to the enrichment of the FLUID models. Furthermore, these techniques offer different services of verification, validation and animation.

3.2 Interactive Systems Development

Model View Controller (MVC). The MVC architecture model [104] is the standard architecture defining the implementation of HMIs introduced in the SmallTalk environment, and it has been adopted by many programming languages afterwards. The Model-View-Controller metaphor is a way to design and implement interactive application software that takes advantage of modularity, in order to help the design of interactive systems, and to allow available pieces already developed for one interactive system to be reused in a new interactive system. The metaphor requires a separation of behaviour between the actual model of the application domain, the views used for rendering the state of the model, and the editing or control of the model and views. The MVC metaphor, illustrated in Figure 3.1, breaks down interactive systems into three components:

- **The model.** The model of an application is the domain-specific software

[5]https://anr.fr/Projet-ANR-16-CE25-0007

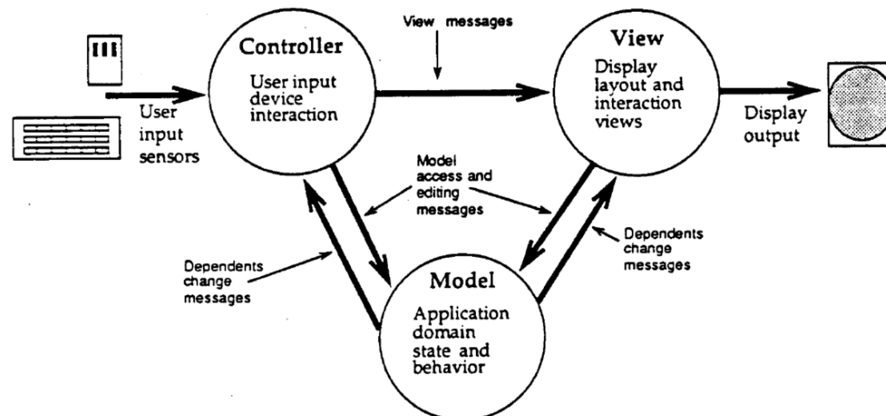


Figure 3.1: MVC architecture model [104]

simulation or implementation of the application's central structure. This can be as simple as an integer (as the model of a counter) or string (as the model of a text editor), or it can be a complex object.

- **The view.** The views provide representation of the model (graphic rendering, sound, ...) ; they request data from their model, and render the data.

- **The controller.** The controllers contain the interface between their associated models and views and the input devices (keyboard, pointing device, time). Controllers also deal with scheduling interactions with other view-controller pairs: they track mouse movement between application views, and implement messages for mouse button activity and input from the input sensor.

The MVC model is a generic architecture model. Different architecture models derived from the standard MVC architecture exist, such as Modified MVC, MVP, and MVVM.

ARINC 661 Standard. ARINC 661 standard [18] defines a standard Cockpit Display System (CDS) interface intended for all types of aircraft installations. The primary objective is to minimise the cost to the airlines, directly or indirectly, by accomplishing the following:

- Minimise the cost of acquiring new avionic systems to the extent it is driven by the cost of CDS development.

- Minimise the cost of adding a new display function to the cockpit during the life of an aircraft.

- Minimise the cost of managing hardware obsolescence in an area of rapidly evolving technology.

- Introduce interactivity to the cockpit, thus providing a basis for airframe manufacturers to standardise the Human Machine Interface (HMI) in the cockpit.

The specification document [18] defines two external interfaces between the CDS and the aircraft systems. The first is the interface between the avionics equipment (user systems) and the display system graphics generators. The second is a means by which symbology and its related behaviour are defined. A User Application is defined as a system that transmits data to the CDS, which in turn can be displayed as visual graphical information to the flight deck crew. A User Application can also include software or hardware that receives input from interactive graphics managed by the CDS.

The CDS provides graphical and interactive services to User Applications within the flight deck environment. When combined with data from User Applications, it should display graphical images to the flight deck crew. This document defines an interface between the CDS and User Applications (UA). The application that controls the interface is defined to be within the CDS. However, this document does not specify the *look and feel* of any graphical information.

In this thesis, specifically in chapter 8, the main emphasis was shed on the widget library specified by ARINC 661 and described in section 3 of the specification standard document. In this section, for each widget the definition is divided into five parts as follows:

- *Definition section* : states the categories of the widget, functional description of the widget and any restrictions to ARINC 661 principles.

- *Widget parameters table* : describes all parameters of the object. These parameters are divided into two categories: ”Commonly used parameters” with a reduced description and ”Specific parameters” with a complete description.

(3) Creation structure table, (4) Event Structure table(s) and (5) Run-time modifiable parameter table give respectively the content and format of the exchanges between UA and CDS, the definition-time exchanges and the run-time exchanges.

Observation (2) The structure and the common requirements of interactive systems are documented and standardised for reuse and sharing. This systems engineering approach proved to be valuable in conventional (not formal) development processes [122, 64]. As a consequence, providing means (ontology modelling language) for formally describing and integrating this domain knowledge or common requirements is beneficial for improving the quality and confidence in the systems formal models.

3.3 The Context of The FORMEDICIS project

This thesis has been achieved in the context of the FORMEDICIS[6] ANR projet which fed the research contributions. Indeed, this thesis addressed the challenge of explicit modelling of domain knowledge which has been identified for interactive critical systems in the context of this project. Nevertheless, this thesis set up the purpose of developing a general framework where domain knowledge may be modelled and the domain-specific requirements may be transferred to systems models independently of the field of application. Indeed, there is ongoing work on explicit modelling of domain knowledge related to autonomous vehicles and railways systems relying on the contributions of the this thesis.

Observation (3). Relying on FORMEDICIS context [119, 151] and the review of formal methods dedicated to interactive critical systems presented in section 3.1, it follows that the major formal methods dedicated to modelling and verifying interactive critical systems lack integrated and systematic framework for explicit modelling of domain knowledge, instead different methodological guidelines and tools are provided to address interactive critical systems modelling and task description and analysis. As a consequence, interactive critical systems represent an adequate source of domain knowledge properties and case studies for supporting the definition of a framework for explicit modelling of domain knowledge.

3.4 Synthesis and Conclusion

The goal of studying works targeting the formal design and verification of interactive critical systems is to identify case studies and relevant domain properties to illustrate the importance of explicit modelling of domain knowledge in the design and verification of such systems. Yet, the framework presented in part II aims at generic application in systems engineering context. Moreover, various formalisms, developed by the community of interactive critical systems, are used to model systems and several relevant properties where many case studies have shown that each formalism has advantages. The criteria for choosing one formalism over another depend to a greater extent on the knowledge and experience of developers in using these formalisms[41]. Different formal techniques of verification are used for achieving verification and validation, such as model checking, equivalence checking, and proof.

On the one hand, the review of major formal methods dedicated to interactive critical systems revealed the relevance of addressing case studies and domain properties pertaining to this category of systems. On the other hand, **observation (1)**, **observation (2)** and **observation(3)** reveals that the issue of explicit modelling of domain knowledge in formal modelling of interactive critical systems will improve the quality and dependability of such systems. Therefore, the chapter provided motivation for

[6] https://anr.fr/Projet-ANR-16-CE25-0007

- the explicit modelling of domain knowledge then transferring the common domain properties to formal models. Examples of such properties are: checking whether a given critical object is always displayed on a specific output device, and checking whether a presentation is redundant (e.g. image and voice in the case of an alarm) in the case of a multi-modal user interface.
- definition of a library of widgets (like ARINC 661 widget library) for reuse and shararability benefits
- addressing behavioural domain properties as *input event leads to triggering a confirmation event* which are required in certification standards.

These challenges are addressed in Part II.

Chapter 4

Case studies

This Chapter contains:

4.1 Traffic Collision Avoidance System 45
4.1.1 Overview of Operation 46
4.1.2 Definitions and Requirements 46
4.2 Multi-Purpose Interactive Application 48
4.2.1 Requirements of WXR User Interface 49
4.3 Automatic Teller Machine 50
4.4 Conclusion 51

This chapter describes case studies borrowed from the field of interactive critical systems. These systems are used throughout the contribution part II to illustrate the framework for explicit modelling of domain knowledge in formal system modelling and the associated methodology. The selection of these case studies was motivated by the adequacy between the properties required for these systems and the demonstration of advantages of the framework: conformance checking, static properties transfer and domain-based behavioural analyses. Another reason for development of multiple case studies is to demonstrate the generality of the framework and its flexibility.

4.1 Traffic Collision Avoidance System

Traffic Collision Avoidance System, or TCAS for short, is used as a case study for comparison between explicit and implicit modelling of domain knowledge in state-based formal methods. It is used in chapter 5 for demonstrating the benefits of explicit modelling of domain knowledge compared to implicit modelling of domain knowledge. Besides, a didactic example manipulating temperatures is used to illustrate the main challenge of explicit modelling of domain knowledge (see section 5.1).

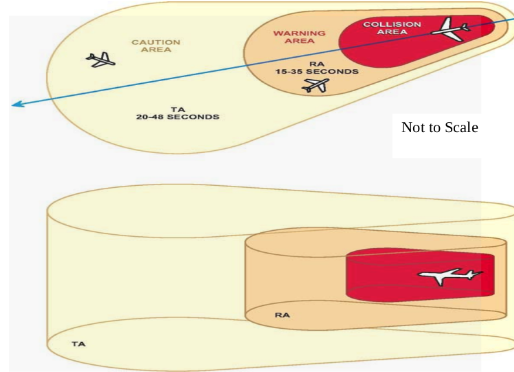


Figure 4.1: TCAS Protection Volume

4.1.1 Overview of Operation

Traffic alert and Collision Avoidance System (TCAS) is an airborne avionics system which operates independently of the ground-based air traffic control (ATC) as a last resort safety net to mitigate the risk of midair collisions. TCAS tracks aircraft in the surrounding airspace through replies from their ATC transponders. If the system diagnoses a risk of an impending collision, it *issues* and *presents* a Resolution Advisory (RA) to the flight crew which guides the pilots to control their vertical rate to avoid a collision [155, 63].

Each TCAS-equipped aircraft is surrounded by a protected volume of airspace. Figure 4.1 shows the horizontal boundaries of the protected volume described using *tau* and *DMOD* (Distance MODification) criteria. The vertical *tau* and the fixed altitude thresholds shall determine the vertical dimensions of the protected volume.

The horizontal dimension of the protected airspace is based on the tau and the approximation of the protected horizontal distance. So, the scale of protected volume depends on the aircraft's speed and trajectory. The horizontal miss distance filter seeks to constrain the protected volume for obtaining sufficient lateral separation using range and bearing information by excluding RAs for aircraft.

Figure 4.2 displays the different symbols used in the traffic display. Shape and colours are used to assist the pilot in understanding the displayed information. Own aircraft is depicted in cyan or white colour and other aircraft are depicted using different geometric symbols and colours as per their level of threat.

4.1.2 Definitions and Requirements

The complete development of the TCAS not necessary to demonstrate the advantages of the framework. Therefore, the main requirements considered in the section chapter 6 are the properties related to the computer-human interface

Figure 4.2: Standardized Symbology for Use on the Traffic Display

of the TCAS: *the property stating that critical interactive aircraft detected in the protected volume are always visible*, which is the safety property of interest. The selection of this case study and this particular property is motivated by its relevance to demonstrate that general domain-specific safety properties (static properties) may be formalised as explicit domain knowledge transferable to a particular system by annotation.

Hereafter, the main definitions and requirements related to the TCAS system informally described in associated standard documents [155, 63, 169] are identified and enumerated. They are structured in three categories: relevant definitions, functional and safety requirements.

Definitions

- **DEF1-Protected Volume:** it is the zone surrounding the protected volume. This volume is subdivided into three layers, from outer to inner: Caution Area (CA_A), Warning Area (WA), and Collision Area (CO_A). In addition, the protected volume is encased in Surveillance Area (SA).

- **DEF2-Critical aircraft:** they are a set of aircraft detected in the zones CA_A, WA or CO_A. Otherwise, the aircraft being out of these zones are considered as not critical.

- **DEF3-Display Grid:** It represents aircraft displayed within the current range (zoom).

- **DEF4-Display Edge:** It represents critical aircraft that are currently out of range and are only partially displayed (half of the symbol).

- **DEF5-Hidden aircraft:** A set of aircraft detected by the TCAS but not displayed.

Functional requirements

The functional requirements associated with the TCAS system are:

- **REQ1-Aircraft Detection:** The primary role of the TCAS is to detect TCAS-equipped aircraft flying in the neighbourhood of own aircraft.
- **REQ2-TCAS zones:** The TCAS zone divides the protected volume into three areas, each corresponding to a different level of criticality for the aircraft.
- **REQ3-Aircraft Displaying:** The second role of the TCAS is to display TCAS-equipped aircraft based on the current range.
- **REQ4-Changing range:** On the display, the TCAS allows to zoom in and zoom out.
- **REQ5-Displaying Range:** Aircraft within the current range are displayed on the TCAS display's grid.
- **REQ6-Symbols Displaying.** Each aircraft is represented using standard symbols and colours.

Safety requirements

The safety requirements associated with the TCAS system are:

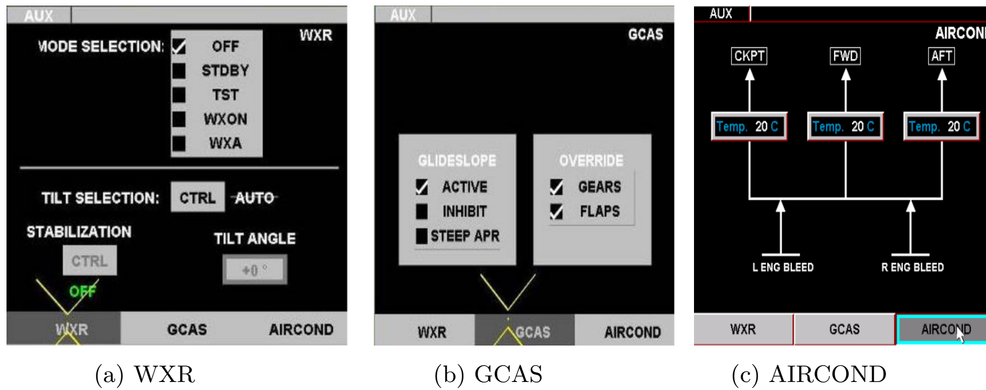
- **REQ7-Unambiguous and complete:** Every detected aircraft must be either within or outside of the range, but never both.
- **REQ8-Display:** Every detected aircraft must either be displayed or hidden, but not both.
- **REQ9-Critical aircraft on screen edge.** When critical aircraft are out of range, they must be displayed on the Screen Edge.
- **REQ10-Visible critical aircraft:** Whatever the range level, critical aircraft must always be visible.

4.2 Multi-Purpose Interactive Application

The Multi-Purpose Interactive Application (MPIA) is an airborne application that meets the ARINC 661 requirements [18]. Fig. 4.3 depicts MPIA, a real User Application (UA) to handle many parameters of the flight. This system provides a tabbed panel with three buttons, WXR for controlling Weather Radar information, GCAS for Ground Collision Avoidance System parameters and AIRCOND for handling air conditioning settings. The crew member can toggle to either application (see Fig. 4.3) by pressing on the corresponding tab. These tabs control three separate programs that can be operated by the pilot and the co-pilot using input system.

The MPIA window in each tab consists of three major parts:

- *information area* is the top bar of every tab that splits into two sections to show the current status of the task on the left-hand side and the error notices, activities in progress or incorrect modification as appropriate on the right.
- *workspace area* depicts adjustments to the selected control panel. For example, the WXR workspace displays all the modifiable parameters of the weather radar system, the GCAS workspace displays some of the operating modes of the GCAS. The AIRCOND workspace displays the selected temperature within the aircraft.
- *menu bar* includes three sections for accessing the WXR, GCAS and AIRCOND digital control panels.



(a) WXR (b) GCAS (c) AIRCOND

Figure 4.3: Snapshots of MPIA

The selection of this case study is motivated by its adequacy to demonstrate conformance checking with respect to a standard. Furthermore, the formal development of the MPIA case study was limited to the computer-human interface associated with the WXR application allowing to control the weather radar information. Indeed, chapter 6 demonstrates the framework for formal conformance checking to prove the conformance of the WXR user interface model with respect to the ARINC 661 standard domain-specific requirements on widgets.

4.2.1 Requirements of WXR User Interface

Hereafter, the requirements of WXR user interface are described, allowing pilots to select different modes from the workspace area and to adjust the orientation (tilt angle) of the weather radar system when necessary. Note that other requirements about GCAS and AIRCOND are omitted.

- **REQ1** There are five working modes:
  - `OFF` for switching off the weather radar,

  - `STDBY` for switching on the weather radar but does not activate the detection,
  - `TST` for displaying graphical test patterns on the radar screen,
  - `WXON` for switching on the weather radar,
  - `WXA` for switching on the radar display and displaying alerts when required.

  Only one mode must be selected at a time.

- **REQ2** The tilt selection mode: AUTO or MANUAL. A CTRL push-button must be allowed to switch between two modes.
- **REQ3** The stabilisation mode: ON or OFF. A CTRL push-button must be allowed to switch between two modes.
- **REQ4** When the tilt selection is in AUTO, the access of CTRL push-button must be forbidden.
- **REQ5** The tilt angle shall be entered within the range [-15: 15] in the edit box.
- **REQ6** When the tilt selection is in AUTO mode then the tilt angle must be unchangeable.

4.3 Automatic Teller Machine

The user interface of an automatic teller machine (ATM) is described hereafter, where the primary requirement is that an authenticated client withdraws banknotes. This case study is used in chapter 7 for demonstrating the a framework for defining and applying domain-based behavioural analyses of formal models.

- **REQ1** A user can exclusively use a keyboard or a screen.
- **REQ2** To withdraw banknotes, a user must be authenticated.
- **REQ3** A user can adjust the brightness a finite number of times.
- **REQ4** Any entered passcode must be followed by a confirmation or an abortion.
- **REQ5** The entered passcode must never be displayed.
- **REQ6** The user may try a new passcode a fixed number of attempts

A user inserts a credit card and chooses an input device to enter a passcode. Upon entering a passcode, the user may confirm it. Before performing this operation, the user may adjust the brightness of the screen. When the user confirms the input, validation starts. It may result in the acceptance or refusal

of the passcode. If the passcode is correct, the ATM delivers banknotes and ejects the card. Otherwise, the user may try again to enter the correct passcode. A user is allowed new attempts a limited and fixed number of times only. A user shall be able to abort the operation of withdrawing banknotes at any time.

4.4 Conclusion

This chapter introduces three case studies that are modelled to demonstrate the effectiveness, flexibility and other advantages of the explicit modelling of domain knowledge in formal methods as presented in the contributions part II. First, the TCAS case study is used to demonstrate that safety properties (static properties) may be defined and proved once and for all and then transferred through annotation to formal models of systems (see chapter 6). Second, the ATM case study is important since it exemplifies the framework for defining and applying domain-specific behavioural analyses on formal models 7. Finally, the MPIA case study is modelled to demonstrate that the framework may be used to achieve conformance checking of formal models regarding general domain-specific requirement predefined in certification standards (see chapter 8). The multiplicity of case studies provides evidence for the utter importance of explicit modelling of domain knowledge in formal modelling and establishes the flexibility of the proposed framework.

Part II

Contributions

The Roadmap of the Contributions

This chapter presents a roadmap for the work carried out during the doctoral thesis for tackling the problem of explicit modelling of domain knowledge in formal modelling. This shows the research trajectory and results achieved since the beginning of the thesis. The doctoral project is related to three scientific disciplines and research areas: *knowledge representation*, *formal methods*, and *interactive systems*. The last one is used as a domain of application for selecting case study requirements and domain properties. Throughout the research, several methodological and technical contributions were proposed to address the main challenges of explicit modelling of domain knowledge in formal modelling. The contributions are presented in the following 4 chapters. The chapters are highlighted and linked to Figure 4.4 with different colours to visualise the interconnections of the contributions.

Figure 4.4 shows the four main contributions to address the challenge of explicit modelling domain knowledge in formal modelling, which are distinguished by three different colours. The relationships between various contributions are also depicted with labelled arrows.

My doctoral research aimed at developing a framework having features identified in the synthesis of section 2.4. They are recalled below:

- **Feature-1** required primitives for modelling domain knowledge concepts and associations preferably generic. This has been addressed by defining a generic data type `Ontology` representing an ontology and its essential components such as concepts, properties, instances and associations. Therefore, representing concepts and associations of some domain knowledge amounts to instantiating this data type.

- **Feature-2** required primitives for expressing constraints of a domain of knowledge. This has been addressed by defining a set of operators entailing these constraints used for modifying or accessing a given ontology.

- **Feature-3** required a mechanism for referencing domain knowledge primitives from design models. This has been addressed simply thanks to the importing mechanism of Event-B theories where the models can use the

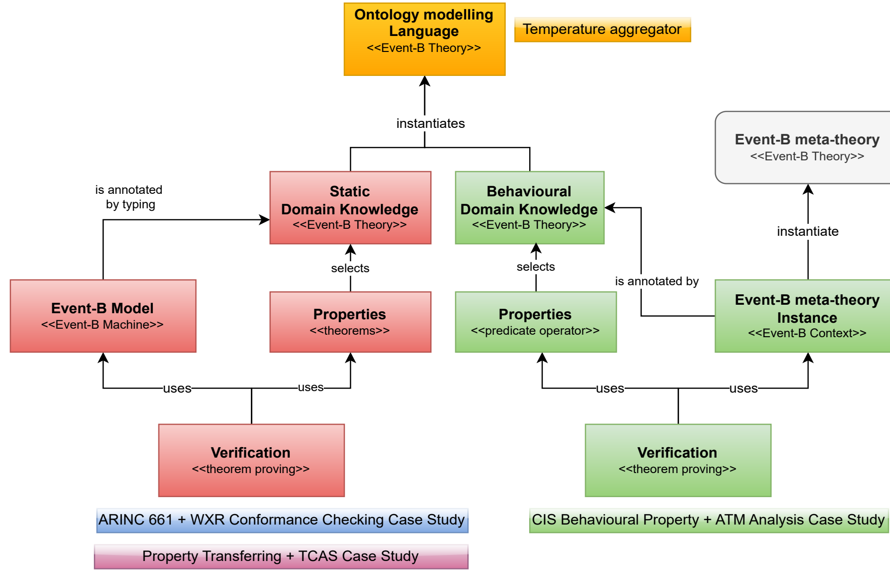


Figure 4.4: A map of the contributions of the thesis

data types and operators of the imported theory. The three first features have been incorporated in the ontology modelling language developed as an generic Event-B theory which is presented in chapter 5.

- **Feature-4** required a mechanism for transferring safety properties (static properties) and behavioural properties (dynamic properties) defined in the domain knowledge to the formal models. This has been addressed for the case of safety properties using typing and one methodological rule requiring from the model to exclusively use the theory primitives. For the case of the behavioural properties analysis, annotation of the events with ontology concepts has been defined and illustrated. The former is presented in chapter 6 and the latter is discussed in chapter 7.

The **Intuition (4)** asserting that standard specification may be viewed as a special case of domain knowledge has been the basis for the development of the contribution presented in the chapter 8.

The **Event-B generic theories** are an essential and underlying component of explicit modelling of domain knowledge framework. Several theories have been defined to support the formal development of explicit domain knowledge. The first step was to define an ontology modelling language as generic Event-B theories for specifying domain knowledge concepts and constraints. It allows the formal description of domain knowledge as Event-B theories. The Event-B models may reference the domain knowledge concepts by using the Event-B theory data types and operators.

The well-definedness feature of the operators of Event-B theories is leveraged for transferring the formalised constraints of domain knowledge. From a type-theoretic point of view, it corresponds to defining dependent types and ensuring that the variables remain well-typed through the evolution of the system. Indeed, the well-definedness condition delimits a part of the sub-domain parameterised by the operator arguments, which is a total function on this sub-domain. A technical presentation of the general approach permitting the simulation of dependent types is discussed and argued in this article [13].

Explicit Modelling of Domain Knowledge

Chapter 5 introduces and presents the work carried out to demonstrate the qualitative leap of explicitly modelling domain knowledge in formal modelling. To this purpose we rely on the paradigm of the triptych, identifying three main parts of the software development process: domain description, requirements prescription and software design. $\mathbf{D}, \mathbf{S} \vdash \mathbf{R}$ expresses a formal deduction, where **D** represents the domain concepts in the form of properties, axioms, relations, functions and theories; **S** represents a system model; and **R** represents the intended system requirements. This entailment states that the given domain description (**D**) and the system model (**S**) yields logically the given requirements (**R**). This advocated paradigm is showcased through a didactic case study (temperature aggregator). An important observation which emerges immediately from this formal modelling methodology is the need for a modelling language to describe domain knowledge. The definition of an ontology modelling language as generic Event-B theory for explicit domain knowledge modelling is discussed in great detail in this chapter. Moreover, this contribution has been published in article [116] where the ontology language is introduced, and its advantages are discussed.

A foundational step towards explicit modelling of domain knowledge is formalising it. Then, formal models refer to domain knowledge concepts, allowing constraints and rules to be transferred. Event-B provides a useful infrastructure for defining such a language by using generic theories. A formal generic theory acting as a meta-model is proposed, and each ontology is described as an Event-B theory that is not necessarily generic but must be an instance of this meta-model. Typically, domain knowledge is formalised as collections of classes, properties and instances.

In Figure 4.4, the yellow component represents the contribution discussed in chapter 5. This part is composed of two boxes, the small box denotes the didactic case study used to demonstrate the benefits of the explicit modelling of domain knowledge, and the large box represents the formal ontology modelling language devised to describe domains of knowledge as ontologies. It is noteworthy that the ontology modelling language serves as the basis for the other contributions.

Transferring Safety Properties

The ontology modelling language defined as a generic Event-B theory in chapter 5 has been used to demonstrate how to transfer safety properties formalised as theorems of a domain theory to formal models of systems. The ontology modelling language which is formalised as an Event-B theory is instantiated to model a domain of knowledge. The domain theory is used in the specification of systems as formal models. A methodology is defined in chapter 6 allowing the transfer of general safety properties mined from domain knowledge to formal models. This methodology is illustrated in a case study borrowed from the field of interactive systems; namely the Traffic Collision Avoidance System (TCAS). Furthermore, the articles [116, 115] presented the benefits of transferring domain properties from domain theories over *ad hoc* specification of such domain properties at the same level as system-specific requirements.

In Figure 4.4, the red part represents the methodology designed for transferring of general domain constraints from domain theories to formal models. It is a common part between the contributions presented in chapter 6 and chapter 8. The difference between the two is highlighted in distinct colours. The main difference is that chapter 8 presents the framework and associated methodology intended for achieving formal conformance checking. The contribution of static annotation for transferring safety properties is highlighted in violet. It is composed of the methodology (red part) and the TCAS model for case study development (violet part)

Analysis of Behavioural Properties

In the figure 4.4, the green colour highlights the contribution presented in chapter 7, where a systematic methodology for specifying behavioural analysis with domain-specific constraints is discussed. This generic framework allows the description and reference of domain knowledge. Indeed, a formal method for describing and setting up domain-specific behavioural analyses is defined and applied to a real-world case study in [118]. The framework supports the definition formal verification technique for dynamic properties entailed by engineering domain knowledge, where Event-B formal models are annotated and analysed in a non-intrusive way. This method is based on the formalisation of behavioural analyses relying on domain knowledge as an ontology on the one hand, and a meta-theory for Event-B on the other hand, introduced in the article [143]. The proposed method is illustrated by analysing the ATM critical interactive system. Chapter 7 describes this contribution, and illustrates it with a useful analysis: investigating that input events are always followed up by confirmation; an important behavioural property commonly raised in the domain of the interactive systems.

Formal Conformance Checking

In Figure 4.4, the blue colour is used to depict the contribution of achieving conformance checking formally. Indeed, the framework for transferring safety properties from domain theories to formal models was leveraged to define a methodology for standard conformance checking. This methodology allows establishing formal conformance accordingly to a standard specification. Therefore, it contributes to enhancing the quality of formal models and increasing the confidence associated with the system under study [117]. In this article, a formal framework based on the Event-B method and its theories is defined for allowing the formalisation of standard concepts and rules as an ontology, especially the formalisation of engineering domains. The description takes the form of an Event-B theory consisting of data types and a collection of operators and theorems. Formal conformance checking is accomplished by annotating the system model with typing conditions. Chapter 8 is dedicated to presenting the formal conformance process and formal models and theories developed for this task. An industrial case study borrowed from the aircraft cockpit engineering domain is used to demonstrate the feasibility and strengths of the approach. The ARINC 661 standard document is formalised using an Event-B theory using the ontology modelling language. This theory formally models, and annotates the safety-critical real-world application of a Weather Radar System for certification purposes.

Chapter 5

Explicit Modelling of Domain Knowledge Using Ontologies

This Chapter contains:

5.1 Temperature Aggregator Example 62
5.1.1 Temperature Aggregator Requirements 62
5.1.2 Modelling without the Theory Operators 63
5.1.3 Modelling with the Theory Operators 64
5.1.4 Synthesis 67
5.2 An Ontology Modelling Language (OML) 68
5.2.1 OML as a Generic Event-B Theory 69
5.2.2 `OntologiesTheory` - Data type 69
5.2.3 `OntologiesTheory` - Operators 69
5.2.4 `OntologiesTheory` - Theorems 73
5.3 Conclusion 73

This chapter is dedicated to demonstrate how explicit modelling of domain knowledge may improve the quality of the formal modelling of systems. Classical modelling refers to all the approaches merging in the same formal specification domain requirements and system requirements. Therefore they are approaches characterised by implicit modelling of domain knowledge. In Event-B, these approaches rely generally on the two original components of Event-B method: the context and the machine components. In contrast, explicit modelling of domain knowledge, as advocated in this thesis, are characterised by a neat separation of the specification of the domain knowledge and the specification of the system. Consequently, these approaches allow for higher-quality modelling, reuse and sharing. Generic Event-B theories are used for separating common requirements related to some domain of knowledge from specific requirements

of the system under design. Event-B generic theories, providing data types and operators with well-definedness, are used with profit to achieve the goal of explicit modelling of domain knowledge in formal modelling.

This chapter is organised as follows. Section 5.1 illustrates the motivation behind the explicit modelling of domain knowledge through two Event-B models of the same system (temperature aggregator): implicit modelling of domain knowledge and explicit modelling domain knowledge cases. Section 5.2 presents the ontology modelling language tailored to explicit modelling domain knowledge, thus providing a uniform framework for expressing and referencing domain concepts and transferring common domain properties. The ontology modelling language opened the path the the contributions presented in chapters 6, 7 and 8. Last, Section 5.3 concludes this chapter.

5.1 Temperature Aggregator Example

This section[1] is dedicated to the demonstration of the advantages of the availability of reusable formal models providing domain knowledge primitives in contrast with a classical modelling approach. The two different approaches are illustrated on an Event-B model of a didactic system: a temperature aggregator. The Event-B method is used to demonstrate the methodology advocated in this thesis. In addition, the approach relies on Event-B theories for formalising and providing domain knowledge models leveraging data types and operators.

The two ways of modelling are illustrated respectively in subsections 5.1.2 and 5.1.3 where domain knowledge on units of temperature is modelled implicitly and explicitly. The interest of double modelling is to pinpoint the differences between the two approaches and highlight the limitations of implicitly embedding domain knowledge in the design models. Finally, a synthesis of explicit and implicit paradigms is presented in Subsection 5.1.4.

5.1.1 Temperature Aggregator Requirements

The case study corresponds to a simple system that computes an average of collected temperature samples. Initially, the average and counter are set to 0. Whenever a new temperature is sensed, the counter is incremented by 1 and the average temperature is updated. The average should be calculated in the same temperature unit as a safety measure. Hereafter, the requirements are enumerated.

- **REQ1** The system should start at 0° Celsius
- **REQ2** The system should compute the average of temperature samples.
- **REQ3** The average should be expressed in Celsius degrees.

[1] The full Event-B modelling of the temperature aggregator example is in appendix C.1

5.1.2 Modelling without the Theory Operators

In this first attempt, the system computing an average of a set of temperatures is modelled using two Event-B components: a context `C_TemperatureContext` in Listing 5.1 and a machine `C_TemperatureMachine` in Listing 5.2. These two components describe domain knowledge and system model respectively. An enumerated set `C_ThermalUnits` defines possible units in `axm1`: `C_Celsius`, `C_Fahrenheit` and `C_Kelvin`. A constant `C_TemperatureT` is defined as a collection of temperature values and units in `axm2`. The constant is a set containing ordered pairs composed of an integer and a unit of the enumerated set. The axiom `axm3` is a necessary constraint on `C_TemperatureT` asserting that the temperatures associated with Kelvin must be non-negative.

```
CONTEXT C_TemperatureContext
SETS C_ThermalUnits
CONSTANTS
  C_Celsius,
  C_Fahrenheit,
  C_Kelvin,
  C_TemperatureT
AXIOMS
  axm1 : partition(C_ThermalUnits, {C_Celsius}, {C_Fahrenheit}, {C_Kelvin})
  axm2 : C_TemperatureT = ℤ × C_ThermalUnits
  axm3 : C_TemperatureT⁻¹[{C_Kelvin}] = ℕ
END
```

Listings 5.1: Temperature aggregator - Event-B context

The machine model is shown in Listing 5.2. Two new variables, `average` and `counter`, are declared (`typing1` and `typing2`). A new safety property is added in (`safetyInv`) to ensure that the unit of average is `C_Celsius`. The `INITIALISATION` event is used to set the initial values of the `counter` and `average`; the two first variables are initialised with 0 and `average` is set to `C_Celcius` (**REQ1**).

```
MACHINE C_TemperatureMachine
SEES C_TemperatureContext
VARIABLES average , counter
INVARIANTS
   typing1 : average ∈ C_TemperatureT
   typing2 : counter ∈ ℕ
   safetyInv : prj2(average) = C_Celsius
EVENTS
   INITIALISATION
   THEN
      act1 : average := 0 ↦ C_Celsius
      act2 : counter := 0
   END
   compute
   ANY newTemperture
   WHERE
      grd1 : newTemperture ∈ C_TemperatureT
      grd2 : prj2(newTemperture) = C_Celsius
   THEN
      act1 : average := (prj1(average) × counter + prj1(newTemperture)) ÷ (counter + 1)
                                                                   ↦ C_Celsius
      act2 : counter := counter + 1
   END
END
```

Listings 5.2: Temperature aggregator - Event-B machine

An event `compute` is added to update the average and counter whenever a new temperature is recorded (**REQ2**). The guard (`grd2`) of this event states that the new sensed temperature is measured in Celsius , and the actions updates `counter` and computes the new `average` (**REQ3**).

Observation. The first attempt exemplifies a common practice in formal modelling, namely, placing domain knowledge together with system requirements, where a developer must encode domain knowledge as axioms and invariants during the system modelling. For example, thermal units are added to axioms, and a safety invariant is added to ensure that the thermal unit is correct for average computation. Therefore the designer has to embed these domain knowledge concepts in the design model and express the safety properties guaranteeing that it is correctly evaluated as an invariant of the model. In this way of modelling, *domain knowledge constraints have to be described for every system model (no reuse).*

5.1.3 Modelling with the Theory Operators

The first attempt highlights that the designer must write invariants properties enforcing the model to entail domain knowledge constraints. An approach to address domain-specific knowledge in system modelling by introducing more structure and separation of concern is presented in this subsection. For this purpose, (1) Event-B generic theories are used to model domain knowledge as a collection of data types, constructors and operators defined by specific axioms. Each operator is accompanied with WD (Well-Definedness) properties defining conditions for correct use of each operator. When an operator is used (i.e. applied), a WD proof obligation, corresponding to this condition, needs to be proved (discharged). A remarkable property ensured by the theory is persevering the same unit between the operands and the result. Next, (2) a model of the aggregator system by using the primitives of the domain theory of temperatures is described.

Theory of Thermal Units

```
THEORY TemperatureTheory
DATA TYPES
    ThermalUnits
        CONSTRUCTORS
            Celsius()
            Fahrenheit()
            Kelvin()

    TemperatureT
        CONSTRUCTORS
            newTemperatureT(value : ℤ, unit : ThermalUnits)
```

Listings 5.3: Event-B theory of temperatures - data type

Listing 5.3 presents an extract from the domain theory for temperatures. Two data types are defined: `ThermalUnits` for enumerating thermal units and `TempertureT` to represent the temperature type made of a value and a unit.

In addition, several operators, such as `tempPlus`, `multByVal`, `divByVal` and so on, are introduced to access and manipulate temperatures (see Listing 5.4). An important predicate operator, `isWDTemperatures`, is defined to ensure that the operands for each manipulated temperature are well-defined meaning that if the unit is `Kelvin` then only positive values are allowed otherwise if the unit is `Celsius` or `Fahrenheit` all values are admitted. Another predicate, `HaveTheSameUnit`, states that a set of temperatures passed in as an argument all have the same unit.

All the defined operators are associated with required well-definednes conditions to guarantee their correctness. For example, adding two temperatures (`tempPlus`) requires that the two temperature parameters are well-defined and have the same unit. Such well-definednes condition is provided in the `tempPlusWD` predicate which is represented by an operator. Similarly, the other operators, `multByVal` and `divByVal` are expressed with their well-definedness conditions, `multByValWD` and `divByValWD`, respectively.

```
OPERATORS
    isWDTemperatures < predicate > (ts : ℙ(TemperatureT))
      well−definedness
        ts ≠ ∅
      direct definition
        ∀t · t ∈ ts ⇒ ((isKelvin(t) ∧ value(t) ≥ 0)
                      ∨((isCelsius(t) ∨ isFahrenheit(t)) ∧ value(t) ∈ ℤ))
    haveTheSameUnit < predicate > (ts : ℙ(TemperatureT))
      well−definedness
        ts ≠ ∅ ∧ isWDTemperatures(ts)
      direct definition
        ∃u · (∀t · t ∈ ts ⇒ (unit(t) = u))
    initTemperatures < expression > ()
      direct definition
        {ts | isWDTemperatures(ts)}
    tempPlusWD < predicate > (t1 : TemperatureT, t2 : TemperatureT)
      direct definition
        isWDTemperatures(t1, t2) ∧ haveTheSameUnit(t1, t2)
    tempPlus < expression > (t1 : TemperatureT, t2 : TemperatureT)
      well−definedness
        tempPlusWD(t1, t2)
      direct definition
        newTemperatureT(value(t1) + value(t2), unit(t1))
    multByValWD < predicate > (t1 : TemperatureT, val : ℤ)
      direct definition
        isWDTemperatures(t1) ∧ val ≥ 0
    multByVal < expression > (t1 : TemperatureT, val : ℤ  )
      well−definedness
        multByValWD(t1, val)
      direct definition
        newTemperatureT(value(t1) × val, unit(t1))
    divByValWD < predicate > (t1 : TemperatureT, val : ℤ)
      direct definition
        isWDTemperatures(t1) ∧ val > 0
    divByVal < expression > (t1 : TemperatureT, val : ℤ)
      well−definedness
        divByValWD(t1, val)
      direct definition
        newTemperatureT(value(t1) ÷ val, unit(t1))
```

Listings 5.4: Event-B theory of temperatures - operators

THEOREMS

$RwSameUnit:$
$$\forall ts \cdot ts \neq \varnothing \wedge \; isWDTemperatures(ts) \wedge haveTheSameUnit(ts)$$
$$\Rightarrow (\exists t1 \cdot t1 \in ts \wedge \forall t \cdot t \in ts \Rightarrow (unit(t) = unit(t1)))$$
$UnionCom:$
$$\forall ts1, ts2, ts3 \cdot (ts1 \neq \varnothing \wedge ts2 \neq \varnothing \wedge ts3 \neq \varnothing \wedge isWDTemperatures(ts1)$$
$$\wedge isWDTemperatures(ts2) \wedge isWDTemperatures(ts3) \Rightarrow$$
$$(haveTheSameUnit(ts1 \cup ts2) \wedge haveTheSameUnit(ts2 \cup ts3)$$
$$\Rightarrow haveTheSameUnit(ts1 \cup ts3)))$$
$\boldsymbol{WDTempPlusThm}:$
$$\forall t1, t2, t \cdot tempPlusWD(t1, t2) \wedge t1 \; tempPlus \; t2 = t \Rightarrow$$
$$isWDTemperatures(t) \wedge haveTheSameUnit(t1, t2, t)$$
$\boldsymbol{WDTempMultThm}:$
$$\forall t1, v, t \cdot multByValWD(t1, v) \wedge t1 \; multByVal \; v = t \Rightarrow$$
$$isWDTemperatures(t) \wedge haveTheSameUnit(t1, t)$$
$\boldsymbol{WDTempDivThm}:$
$$\forall t1, v, t \cdot divByValWD(t1, v) \wedge t1 \; divByVal \; v = t \Rightarrow$$
$$isWDTemperatures(t) \wedge haveTheSameUnit(t1, t)$$

Listings 5.5: Event-B theory of temperatures - theorems

In addition, two important theorems have been defined (see `THEOREMS` clause presented in Listing 5.5 to express properties associated with the defined operators. Here, the property formalised is that applying the operators on well-defined temperatures also yields well-defined temperatures. The theorems **WDTempPlusThm**, **WDTempMultThm** and **WDTempDivThm** assert that the operators `tempPlus`, `multByVal` and `divByVal` respectively entail the property that their results are a well-defined temperature. Other data types and operators can be defined to model additional knowledge related to temperature manipulation.

A Model for Computing the Average Temperature

Temperature theory (`TemperatureTheory`) is used to develop the temperature aggregator model. Listing 5.6 shows an Event-B machine that models the temperature aggregator system using the defined types and operators.

Two state variables are declared in this machine, *average* and *counter*, which are of type `TemperatureT` and $\mathbb{N}$, respectively. Moreover, we introduce an additional typing invariant and a theorem (`SafetyThm`) to ensure that the required safety properties and WD conditions associated with the defined operators. The theorem asserts that the `average` variable is always a well-formed temperature. We set initial values for each declared variable in `INITIALISATION` event. A new event, *computeAvg*, is defined to calculate the average temperature using our defined theory operators. The event's guard states that any sensed temperature is typed as $TemperatureT$ and is well-defined ($grd1$-$grd2$). The last guard checks that the *average* and *newTemperature* have the same unit. The actions of the event are used to update *average* and *counter* variables. Note that the average calculation only employs theory-defined operators.

In this development, a set of proof obligations related to well-definedness conditions associated with theory operators (e.g. `tempPlus` and `multByVal`) representing safety properties is generated. In particular safety properties requiring that the temperatures remain well-defined and have the same unit need

to be discharged. However, the proof is straightforward thanks to the more general and universally quantified theorem in `TemperatureTheory` and the working hypothesis requiring that only the data types and operators shall be used in the system modelling.

```
MACHINE
    T_TemperatureMachine
VARIABLES    average,
             counter
INVARIANTS
  Typing1 : average ∈ TemperatureT
  Typing2 : counter ∈ ℕ
  UsedOP : average ∈ initTemperatures)∨
        (∃avg, nt · tempPlusWD(avg, nt) ∧ average = avgtempPlus nt)∨
        (∃avg, c · multByValWD(avg, c) ∧ average = avg multByVal c)∨
        (∃avg, c · divByValWD(avg, c) ∧ average = avg divByVal c)
THEOREMS
  SafetyThm : isWDTemperatures(average)
EVENTS
    INITIALISATION
  THEN
      act1 : average := newTemperatureT(0, Celsius)
      act2 : counter := 0
  END

   computeAvg
  ANY   newTemperature
  WHERE
      grd1 : newTemperature ∈ TemperatureT
      grd2 : isWDTemperatures({newTemperature})
      grd3 : haveTheSameUnit(average, newTemperature)
  THEN
      act1 : average := ((average multByVal counter)
                         tempPlus newTemperature) divByVal (counter + 1)
      act2 : counter := counter + 1
  END
END
```

Listings 5.6: Theory-based temperature aggregator - Event-B machine

5.1.4 Synthesis

Previously, two models of the same system were presented. The first model can be qualified as monolithic and the second model may be described as modular. The first model includes the requirements of the system and the common domain knowledge in a single specification. Therefore constraints related to the domain of temperatures like the homogeneity of the operations (the temperatures must have the same unit for coherent computation) are specified as specifics of the system. As a consequence, this approach has several limitations in several ways: *reusability* and *sharability*, and *validation by expertise*

The second model overcomes the limitations of the previous one. First, the domain knowledge is formalised once and for all in a single *reusable* and *shareable* theory, and second, the system modelling is simplified since it is no longer necessary to write invariants or properties related to domain knowledge to guarantee correct computation of average temperatures. Additionally, Event-B theories enable the design of models to comply with theories. Indeed, theorems of the used theory are equally theorems of the models provided that the models

use only the operators associated with the data type or a composition of these operators.

`UsedOP` invariant (see Listing 5.6) expressing the condition that the only operators allowed by the method are those provided by the theory. More precisely the invariant embeds the fact that the reachable states are only those obtainable by the theory operators or their combination. It is a prerequisite to transfer the properties established for the domain theory to the models.

Induction is used for establishing that the state variable typed by the theory data type (`average` in our example) belongs to the safety domain. Note that the proof of `UsedOP` requires induction, especially the hypothesis stating that the state variable is located in the safety zone which is ensured by the use of the operators provided by the theory (`tempPlus`, `multByVal` and `divByVal` for the theory of temperatures). An essential point of applying this methodology is that the domain properties are transferrable from the theory to the models. Indeed, the proof process is simplified since the theorem **SafetyThm** is discharged straightforwardly using the theorems of the theory.

Note. Event-B theories support the description of other operators like `tempPlus` and `multByVal` which use arithmetic. Therefore, depending on the chosen Ontology Modelling Language (**OML**), Event-B theories permit the modelling of complex domain knowledge. However, such theories may require interactive proof reasoning instead of fully automatic. The choice of the OML is driven by the needs and complexity of the domain knowledge of interest, in our case system engineering. This is discussed more thoroughly in chapter 2.

5.2 An Ontology Modelling Language (OML)

Subsection 5.1.4 concluded by emphasising the importance of the *explicit* representation of the domain knowledge in order to make modelling more reusable and sharable. However, the classical modelling style of integrating the domain knowledge may lead to heterogeneous descriptions which may jeopardise or at least limiting these two goals. Therefore a common language for describing domain knowledge is necessary. In other words, an ontology modelling language (OML) (see section 2.2) is an important step toward the goal of providing a framework for describing domain knowledge that can be used in formal system modelling with reusability and sharability. Indeed, it needs to be a high-level language capable of encapsulating domain knowledge as well as domain-specific properties for complex systems. On the one hand, it must be expressive enough, while on the other hand, it needs to provide high-level modelling constructs for an easy description of domains. First, the expressiveness aspect is guaranteed by the fact that Event-B is based on typed set theory and first-order logic. Second, Event-B theories are an effective way for allowing the definition of high-level constructs by using algebraic specification composed of data types, operators and theorems, and proof rules.

5.2.1 OML as a Generic Event-B Theory

An ontology modelling language is proposed based on a study of two ontology description languages: OWL and PLib (see subsection 2.2.3). This subsection [2] is dedicated to the presentation of a novel ontology modelling language which is formalised as generic Event-B theory `OntologiesTheory`. In the following, different subsections discusses the structure of this ontology modelling language: data type (see subsection 5.2.2), operators (see subsection 5.2.3), and theorems (subsection 5.2.4).

5.2.2 `OntologiesTheory` - Data type

Formalising domain knowledge requires first to provide concepts, properties, and instances. In `OntologiesTheory`, three parameters are defined: `C`, `P`, and `I` for classes, properties, and instances, respectively. The data type `Ontology(C,P,I)` has one generic constructor `consOntology` which requires 7 arguments. `classes`, `properties`, `instances` describe classes, properties and instances respectively of the ontology being constructed. Then, `classAssociations` contains the relation between classes and `classProperties` assigns properties to classes.

THEORY $OntologiesTheory$
TYPE PARAMETERS C, P, I
DATA TYPES
$Ontology(C, P, I)$
CONSTRUCTORS
$consOntology(classes : \mathbb{P}(C), properties : \mathbb{P}(P), instances : \mathbb{P}(I),$
$classProperties : \mathbb{P}(C \times P),$
$classInstances : \mathbb{P}(C \times I),$
$classAssociations : \mathbb{P}(C \times P \times C),$
$instanceAssociations : \mathbb{P}(I \times P \times I))$

Listings 5.7: `OntologiesTheory` - data type

Next, `ClassInstances` is a function that gives every class the set of its instances. Finally, `InstanceAssociations` is an essential component defining the relationships between the instances of the ontology. `classAssociations` and `InstanceAssociations` are closely related since the former dictates the structure of the latter. `InstanceAssociations` defines that an ontology instance is related to another instance via some property only if the class of the first instance is also related to the class of the last instance via the same property (see definition of `isWDInstancesAssociations` in Listing 5.8).

5.2.3 `OntologiesTheory` - Operators

The definition of the ontology structure via the data type `Ontology(C,P,I)`, the `OntologiesTheory` provides a collection of operators to safely manipulate the data type instances. These operators are defined by providing direct definitions and WD conditions to express desired properties and functionalities. WD conditions associated with each operator ensures the correct use and the

[2] The full listing of `OntologiesTheory` is in appendix A.1

preservation of a valid ontology structure during instantiation. For example, `getInstanceAssociations` is an operator that returns the association between instances if it complies with the relation between classes. To use this operator correctly we need to ensure that the `isWDInstanceAssociations` predicate holds. It states that the relation between instances is compatible with the relation between classes as defined in `classAssociations` (see definition of `isWDInstancesAssociations` in Listing 5.8). i. e. two instances are related by a property only if their classes are related with the same property.

```
OPERATORS
 getClasses < expression > (o : Ontology(C, P, I))
  direct definition
    classes(o)
 getProperties < expression > (o : Ontology(C, P, I))
  direct definition
    properties(o)
 getInstances < expression > (o : Ontology(C, P, I))
  direct definition
    instances(o)
 isWDClassProperites < predicate > (o : Ontology(C, P, I))
  direct definition
    classProperties(o) ∈ getClasses(o) ↔ getProperties(o)
 getClassProperties < expression > (o : Ontology(C, P, I))
  well−definedness isWDClassProperites(o)
  direct definition
    classProperties(o)
 isWDClassInstances < predicate > (o : Ontology(C, P, I))
  direct definition
    classInstances(o) ∈ getClasses(o) ↔ getInstances(o)
 getClassInstances < expression > (o : Ontology(C, P, I))
  well−definedness isWDClassInstances(o)
  direct definition
    classInstances(o)
 isWDClassAssociations < predicate > (o : Ontology(C, P, I))
  well−definedness isWDClassProperites(o)
  direct definition
    classAssociations(o) ∈ getClassProperties(o) → classes(o)
 getClassAssociations < expression > (o : Ontology(C, P, I))
  well−definedness isWDClassAssociations(o)
  direct definition
    classAssociations(o)
 isWDInstancesAssociations < predicate > (o : Ontology(C, P, I))
  well−definedness
      isWDClassProperites(o) ∧ isWDClassInstances(o) ∧ isWDClassAssociations(o)
  direct definition
    instanceAssociations(o) ⊆ instances(o) × properties(o) × instances(o)∧
    instanceAssociations(o) ⊆ {i1 ↦ p ↦ i2 |
        i1 ∈ I ∧ p ∈ P ∧ i2 ∈ I∧
        i1 ↦ p ↦ i2 ∈ instances(o) × properties(o) × instances(o)∧
        (∃c1, c2 · c1 ∈ C ∧ c2 ∈ C ∧ {c1, c2} ⊆ getClasses(o) ⇒
               (c1 ↦ p ↦ c2 ∈ getClassAssociations(o)∧
               p ∈ getClassProperties(o)[{c1}]∧
               i1 ∈ getClassInstances(o)[{c1}]∧
               i2 ∈ getClassInstances(o)[{c2}]))}
 getInstanceAssociations < expression > (o : Ontology(C, P, I))
  well−definedness isWDInstancesAssociations(o)
  direct definition
    instanceAssociations(o)
 isWDOntology < predicate > (o : Ontology(C, P, I))
  direct definition
    isWDClassProperites(o) ∧ isWDClassInstances(o)∧
    isWDClassAssociations(o) ∧ isWDInstancesAssociations(o)
```

Listings 5.8: `OntologiesTheory` - basic accessor operators

In the same vein, the operators for accessing the ontology are associated with predicated operators formalising the well-definedness conditions. For example, `isWDClassAssociations` describes the WD condition of `getClassAssociations` stating that the associations are valid when they relate classes (source and target) already provided via valid properties of the source classes. To improve readability, a good practice is well-definedness conditions of a given operator are defined as a single predicate whose name is prefixed with `isWD`.

```
ontologyContainsClasses < predicate > (o : Ontology(C, P, I), cc : ℙ(C))
   well−definedness isWDOntology(o) ∧ cc ≠ ∅
   direct definition
     cc ⊆ getClasses(o)
ontologyContainsProperties < predicate > (o : Ontology(C, P, I), pp : ℙ(P))
   well−definedness isWDOntology(o) ∧ pp ≠ ∅
   direct definition
     pp ⊆ getProperties(o)
ontologyContainsInstances < predicate > (o : Ontology(C, P, I), ii : ℙ(I))
   well−definedness isWDOntology(o), ii ≠ ∅
   direct definition
     ii ⊆ getInstances(o)
ontologyContainsIpv < predicate > (o : Ontology(C, P, I), ipvs : ℙ(I × P × I))
   well−definedness isWDOntology(o)
   direct definition
     ipvs ⊆ getInstanceAssociations(o)
```

Listings 5.9: `OntologiesTheory` - basic tester operators

`isWDOntology` is a key operator that serves as a prerequisite for all other operators; it is part of the WD of any operator manipulating an ontology. All operators must be applied to a valid ontology to produce meaningful results (valid ontology). The defined predicate operator ensures the validity of an ontology. It is a conjunction of 4 main terms ensuring that all the ontology's components are well-built: `isWDClassProperties`, `isWDClassInstances`, `isWDClassAssociations` and `isInstancesAssociations`. The two last predicates have previously been explained. `isWDClassProperties` and `isWDClassInstances` signify that properties and instances assigned to the classes in a given ontology have already been specified in the `properties` and `instances` respectively.

```
getInstancesOfaClass < expression > (o : Ontology(C, P, I), c : C)
   well−definedness isWDOntology(o) ∧ ontologyContainsClasses(o, {c})
   direct definition
     getClassInstances(o)[{c}]
getValueOfAInstanceProperty < expression > (o : Ontology(C, P, I), i : I, p : P)
   well−definedness isWDOntology(o) ∧ ontologyContainsProperties(o, {p})
     ∧ontologyContainsInstances(o, {i})
   direct definition
    getInstanceAssociations(o)[{i ↦ p}]
getClassesOfInstance < expression > (o : Ontology(C, P, I), i : I)
   well−definedness isWDOntology(o) ∧ ontologyContainsInstances(o, {i})
   direct definition
     getClassInstances(o)⁻¹[{i}]
```

Listings 5.10: `OntologiesTheory` - other accessor operators

Additionally, the ontology theory provides a set of predicate operators for testing that ontologies contain elements such as classes, properties or instances.

For example, `OntologyContainsClasses` allows one to check if a given class is part of the ontology. The Listing 5.9 enumerates several testing predicate operators for testing the composition of a given ontology.

Listing 5.10 shows several expression operators for requesting computed attributes of a given ontology. For example, `getInstancesOfaClass` returns all the instances of a given class. Another example, `getPropertyRangeClasses` operator returns all the target classes that are involved in class associations for a given property. They are important for defining test predicate operators (see Listing 5.10).

The ontology theory provides other operators for checking the content of an ontology like verifying whether a given instance is related to another instance via a given property. This special test is provided by `instanceHasPropertyValuei` operator. `CkeckOfSubsetOntologyInstances` is a key operator it allows checking whether a schema of instances meaning a collection of triple instance, property and instance is a subset of the allowed combination in the sense of `isWDInstancesAssociations`. In Listing 5.11, different operators serving to check this kind of composition properties are enumerated.

```
classContainsInstances < predicate > (o : Ontology(C, P, I), c : C, ii : ℙ(I))
  well−definedness isWDOntology(o) ∧ ontologyContainsClasses(o, {c})∧
  ontologyContainsInstances(o, ii)
  direct definition
    ii ⊆ getInstancesOfaClass(o, c)
instanceHasPropertyValue < predicate > (o : Ontology(C, P, I), i : I, p : P, v : I)
  well−definedness isWDOntology(o) ∧ ontologyContainsInstances(o, i, v)
      ontologyContainsProperties(o, p)
  direct definition
    v ∈ getValueOfAInstanceProperty(o, i, p)
CkeckOfSubsetOntologyInstances
                  < predicate > (o : Ontology(C, P, I), ipvs : ℙ(I × P × I))
  well−definedness isWDOntology(o)
  direct definition
    ipvs ⊆ getInstanceAssociations(o)
isWDInstanceHasPropertyValuei < predicate > (o : Ontology(C, P, I),
                  ipvs : ℙ(I × P × I), i : I, p : P, v : I)
  direct definition
    isWDOntology(o) ∧ CkeckOfSubsetOntologyInstances(o, ipvs)
    ∧ontologyContainsInstances(o, {i, v}) ∧ ontologyContainsProperties(o, {p})
instanceHasPropertyValuei < predicate > (o : Ontology(C, P, I),
                  ipvs : ℙ(I × P × I), i : I, p : P, v : I)
  well−definedness isWDInstanceHasPropertyValuei(o, ipvs, i, p, v)
  direct definition
   v ∈ ipvs[{i ↦ p}]
```

Listings 5.11: `OntologiesTheory` - other tester operators

```
isA < predicate > (o : Ontology(C, P, I), c1 : C, c2 : C)
  well−definedness isWDOntology(o) ∧ ontologyContainsClasses(o, {c1, c2})
  direct definition
    getInstancesOfaClass(o, c1) ⊆ getInstancesOfaClass(o, c2)
```

Listings 5.12: `OntologiesTheory` - `isA` operator

Moreover, `OntologiesTheory` provides the `isA` operator, that formalises the subsumption relationship between classes. It allows testing whether a class is a subclass of a given class. Formally, it is defined in terms of instances where

a class c_1 is said to subsume a class c_2 if and only if the instances of c_2 are included in the instances of c_1.

5.2.4 `OntologiesTheory` - Theorems

In this theory, we also introduce new theorems based on the defined operators that may aid in the development and proof of system models.

```
THEOREMS
  isATrans :
  ∀o, c1, c2, c3 · o ∈ Ontology(C, P, I) ∧ isWDOntology(o) ∧ c1 ∈ C ∧ c2 ∈ C ∧ c3 ∈ C∧
                    ontologyContainsClasses(o, {c1, c2, c3})
                        ⇒ (isA(o, c1, c2) ∧ isA(o, c2, c3) ⇒ isA(o, c1, c3))
  unionCompt :
   ∀o, cs1, cs2 · o ∈ Ontology(C, P, I) ∧ isWDOntology(o)∧
          cs1 ⊆ C ∧ cs2 ⊆ C ∧ cs1 ≠ ∅∧
          cs2 ≠ ∅ ∧ ontologyContainsClasses(o, cs1) ∧ ontologyContainsClasses(o, cs2)
                  ⇒ (ontologyContainsClasses(o, cs1 ∪ cs2))
```

Listings 5.13: `OntologiesTheory` - theorems

Listing 5.13 contains two important theorems, **isATrans** and **unionCompt**, the first asserts that the `isA` is transitive, and the second states that `ontologyContainsClasses` is compatible with the set union. Note that these properties are valid as long as the ontology is well-defined; `isWDOntology` is a hypothesis of the theorems.

5.3 Conclusion

This chapter addressed the challenge of explicit modelling of domain knowledge and constraints associated to systems. The chapter presented the motivation behind the advocacy of explicit modelling of domain knowledge and illustrated the differences regarding the implicit modelling alternative. For this purpose, a didactic case study of temperature aggregator has been used to illustrate this two modelling approaches. First, the explicit modelling allows formalising the environment and domain requirements related to systems, therefore safeguarding the system development from false or incomplete domain requirements. Then, the availability of a general formal specification of domain requirements fosters the reuse and the share of common requirements across models of different systems. Next, referencing primitives of formal specification of a domain enables the transfer of predefined domain requirements like those specified in various standards documents usually in natural language. After that, the neat separation of concerns arising when specifying domain-specific requirement and system-specific requirements facilitates the inspection of different parts and validation by experts. Finally, the proof and modelling efforts are greatly mitigated since theorems specification and proof of common domain-requirements is carried out once and for all.

The second section of this chapter was dedicated to the discussion of the importance of a common formal language for describing domain-specific concepts and constraints. Consequently, an ontology modelling language was presented

as a generic Event-B theory where a generic data type was defined, and a collection of operators and theorems are described. This language plays an essential role in the definition of different domain ontologies, useful for achieving the contributions presented in chapters 6, 7 and 8.

Chapter 6

Annotation-Based Transfer of Safety Properties

This Chapter contains:

6.1 Our Approach 76
 6.1.1 Generic Part: the definition of the Domain Theory 77
 6.1.2 Specific Part: Annotating the System 77
6.2 TCAS Case Study 77
 6.2.1 An Ontology of Interactive Objects 78
 6.2.2 Instantiation of the Displayability Theory 84
 6.2.3 Modelling without the Theory Operators 85
 6.2.4 Modelling with the Theory Operators 87
 6.2.5 Proof Statistics 89
6.3 Conclusion 90

This chapter is dedicated to the presentation of the formal framework that enables system modelling to refer to domain knowledge models, permitting the domain knowledge safety properties to be transferred to these system models. The behavioural properties are not considered in this chapter; they are investigated in chapter 7. The approach is based on an annotation of the design models: the variables are typed with the domain knowledge concepts and manipulated with custom operators yielding the properties to be enforced on the models. The ontology modelling language allows defining the domain knowledge concepts and constraints which are then enforced on the design model through the annotation mechanism.

The chapter is organised as follows. Section 6.1 presents the framework for annotation enabling safety properties transfer from domain theories to formal models using the ontology modelling language. Section 6.2 illustrates the framework and the methodology presented in this chapter using a real-world system: the Traffic Collision Avoidance System (TCAS). Section 6.3 concludes this chapter.

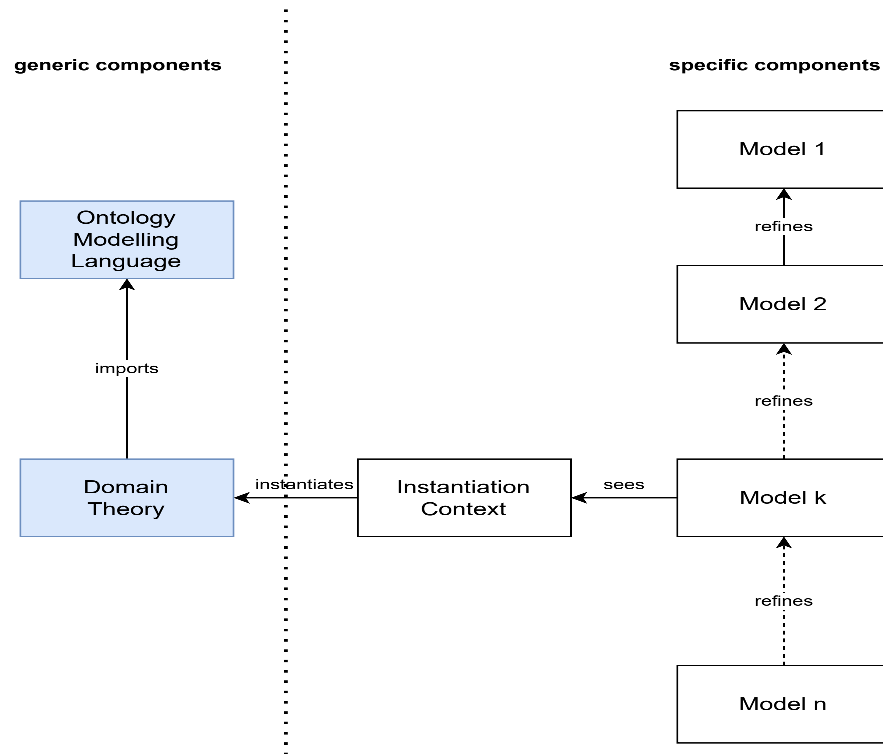


Figure 6.1: The framework for modelling with explicit domain knowledge

6.1 Our Approach

In this section, the framework allowing the definition of domain knowledge concepts and properties is presented and a mechanism of annotation for safety property transfer is discussed. The structure of the framework is shown in Figure 6.1; it has two main parts: *Generic* and *Specific*. The goal of developing generic components is to define them once and for all, while specific components can be defined for the specific system by instantiating the generic components. When developing generic components, the ontology modelling language defined in Chapter 5 is used. Indeed, the ontology modelling language offers primitives for expressing the domain knowledge concepts and associations (see Section 5.2). It is the basis for all domain knowledge theories, like interactive objects (see section 6.2.1) , ARINC 661 (see section 8.3) or events (see section 7.4.1) domain ontologies. Domain theories can be described using this ontology modelling language. In practice, operators need to be defined to convey the desired properties of domain knowledge which are formalised and proved explicitly as theorems.

6.1.1 Generic Part: the definition of the Domain Theory

The first step, in transferring safety properties from domain theories to formal models, is to define the relevant domain theories. The definition of domain theories is achieved using the ontology modelling language presented in section 5.2. This fact is illustrated, in Figure 6.1, by the domain theory importing the ontology modelling language theory. The description of the domain theory comprises the definition of domain safety properties as theorems. The definition of theorems has at least two benefits. First, it is a way to express clearly and unambiguously the important safety properties intended for some domain of interest. Secondly, if a system model is described methodically using a combination of these operators then it may straightforwardly be proven that the model complies with the domain knowledge rules formalised as theorems in a theory. Indeed, the conformance of a system model to a corpus of knowledge is a decisive requirement in system engineering. This process is commonly called system certification according to standards.

6.1.2 Specific Part: Annotating the System

Once the domain theory is defined, it may be used in formal modelling. The domain theory is instantiated and referenced by the system models. The main objective of developing specific components of our framework is to formalise the system's special features and behaviour. A formal model of a system can be described using the elements of the domain theory, namely the data types and operators. The specific part of the proposed methodology allows for incremental modelling of the system, with the domain theory data types and operators referenced in the formal description. It is composed of a model refinement chain consisting of Event-B contexts and Event-B machines. Another component that must be provided at each system development is the instantiation of the domain theory for specifying the information specific to the system because this is where the components of the ontology data type related to instances are defined.

6.2 TCAS Case Study

In this section[1], the real-world system is modelled in accordance with the framework previously defined. It demonstrates how domain properties are transferred from the domain theory to the system model. The informal description of the Traffic Collision Avoidance System (TCAS) is provided in section 4.1.

The case study is developed in two versions to showcase the importance of the framework related to annotation for safety properties transfer compared to hard-coding these safety properties within the formal models. Subsection 6.2.3 presents the ad hoc modelling used to specify the safety properties in the formal model, and subsection 6.2.4 shows an improved version of the formal model

[1] The full Event-B development of TCAS case study is in appendix C.2

where the domain safety properties are formalised explicitly and are referenced from the domain theory.

The domain of interest for the development of the TCAS is interactive objects. This domain is described as an ontology by instantiating the ontology modelling language `OntologiesTheory` where the key concepts are visibility and criticality. The visibility of an object represents the status of this object whether displayed or not, and the criticality of an object represents the level of threat with respect to the system of interest. Subsection 6.2.1 presents important parts of the Event-B theory of interactive objects `DisplayabilityTheory` formalised as an ontology.

6.2.1 An Ontology of Interactive Objects

`DisplayabilityTheory`[2] is formalised as an ontology therefore it instantiates the ontology theory. The concepts and the properties of the ontology of interactive objects are defined in an axiomatic way. This subsection discusses the three main clauses of `DisplayabilityTheory`

Constants and Operators Declaration

The ontology of interactive objects needs to provide concepts, properties and instances to instantiate the `OntologiesTheory`.

```
AXIOMATIC DEFINITIONS   IOOntologyAxiomatisation :
TYPES
    IOClasses,
    IOProperties,
    IOInstances
OPERATORS
    criticality < expression > () : IOClasses
    visibility < expression > () : IOClasses
    hasVisibility < expression > () : IOProperties
    hasCriticality < expression > () : IOProperties
    visible < expression > () : IOInstances
    hidden < expression > () : IOInstances
    critical < expression > () : IOInstances
    safe < expression > () : IOInstances
```

Listings 6.1: `DisplayabilityTheory` - constants

Listing 6.1 declares the concepts and properties involved in the description of interactive objects. Note that these declarations are written in the `OPERATORS` clause of the Event-B theory but they are constants which are used to populate the three main abstract types needed to instantiate the ontology theory. Indeed, the instantiation requires providing three abstract types `IOClasses`, `IOProperties`, and `IOInstances`: they represent the set of classes, properties and instances of the ontology respectively. The theory then defines many constants, typed with theory's abstract types. It includes `criticality`, `visibility` as elements of `IOClasses` which represent the main concepts of the ontology of interactive objects, and `critical`, `safe`, `visible`, `hidden` as

[2]The full listing of `DisplayabilityTheory` is in appendix B.1

elements of `IOInstances` representing instances of the ontology of the interactive object. The attributes of interactive objects are defined as well, and they are typed with `IOProperties`; we defined `hasVisibility` and `hasCriticality` properties. The definition of the types `IOClasses`, `IOProperties`, and `IOInstances` as defined with axioms in the `AXIOMS` clause (see Listings 6.4, 6.5 and 6.6).

```
isIOOntologyWD < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances)) :
isVisibleiWD < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances) :
isVisiblei < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances), ipvs :
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances) :
well−definedness isVisibleWDi(o, ipvs, i)
isHiddeniWD < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
isHiddeni < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
well−definedness isHiddenWD(o, ipvs, i)
isCriticaliWD < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
isCriticali < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
well−definedness isCriticalWD(o, ipvs, i)
isSafeiWD < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
isSafei < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
well−definedness isSafeWD(o, ipvs, i)
isWDSetVisiblei < predicate >
        (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
```

Listings 6.2: `DisplayabilityTheory` - tester operators

In addition, the ontology of interactive objects must provide several operators to manipulate the ontology to ensure that the manipulation preserves the domain safety properties. Therefore, many operators are defined to update and check properties on an instance variable. A key predicate operator is `isIOOntologyWD` predicate that holds when the ontology passed as an argument is well-defined in a sense that is defined in the `AXIOM` clause. Moreover, operators for verifying particular facts on an ontology of interactive objects are defined. `isVisiblei`, `isHiddeni`, `isCriticali`, and `isSafei` are predicate checking that a given instance is visible, hidden, critical, and safe respectively with respect to the ontology passed in as an argument. Note that these operators are associated with well-definedness conditions formalised as predicates operators. For example, the well-definedness conditions of `isCriticali` are grouped under the operator `isCriticaliWD`. The definitions of these operators are equally formalised as axioms in the `AXIOM` clause. All these operators take three parameters: an instance of `Ontology(IOClasses, IOProperties, IOInstances`, a collection of triples `ipv` which represent generally the single state variable representing the interactive critical system, the last parameter represent the interactive object to be checked.

Additionally, expression operators updating a part of the ontology specifically the instances associations are provided, as well, in accordance with the encapsulation principle. Event-B doesn't provide any built-in encapsulation principle as in other modelling languages like UML or object-oriented program-

ming languages like Java or C++ classes, and Ada packages, etc. This principle is embodied in the methodology of using the theory, i.e., the models are required to use exclusively the operators provided by the theory to benefit from the domain safety properties they entail.

```
isWDSetVisiblei < predicate >
        (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
setVisiblei < expression > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances),
        i : IOInstances) :
        ℙ(IOInstances × IOProperties × IOInstances)
well−definedness  isWDSetVisiblei(o, ipvs, i)
isWDSetHiddeni < predicate >
        (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
setHiddeni < expression > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances),
        i : IOInstances) :
        ℙ(IOInstances × IOProperties × IOInstances)
well−definedness  isWDSetHiddeni(o, ipvs, i)
isWDSetSafei < predicate > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances), i : IOInstances)
setSafei < expression > (o : Ontology(IOClasses, IOProperties, IOInstances),
        ipvs : ℙ(IOInstances × IOProperties × IOInstances),
        i : IOInstances) :
        ℙ(IOInstances × IOProperties × IOInstances)
well−definedness  isWDSetSafei(o, ipvs, i)
```

Listings 6.3: `DisplayabilityTheory` - accessor operators

Listing 6.3 presents an extract of a collection of expression operators provided by the ontology of interactive object theory. As stated previously, the operators are associated with well-definedness conditions which are formalised, and grouped under a single predicate operator. For example, the well-definedness conditions of `setHiddeni` are formalised in `isWDSetHiddeni`. The axioms state that the classes of this ontology include `visibility` and `criticality`. The types `IOProperties`, `IOInstances` are populated as well.

Axioms Definition

The definitions of different abstract types and operators are given in the axioms clause. Listings 6.4, 6.5 and 6.6 present the definitions of the types, predicates and expression operators respectively, which are declared and commented previously. The first axioms (see Listing 6.4) populate the abstract types `IOClasses`, `IOProperties`, `IOInstances` which define the building blocks of the ontology of interactive objects.

```
AXIOMS
    IOClasses : {visibility, criticality} ⊆ IOClasses
    IOProperties : partition(IOProperties, {hasVisibility}, {hasCriticality})
    IOInstances : {visible, hidden, critical, safe} ⊆ IOInstances
```

Listings 6.4: `DisplayabilityTheory` - axioms for types

The definition of the predicate operators and well-definedness conditions are presented in Listing 6.5. The `isIOOntologyWD` predicate is built based on the

predicate operator `isWDOntology` specifying that an ontology is well-defined (see Listing 5.8). Other conditions enrich the definition of this operator; specifically, a well-defined interactive ontology shall include the instances, classes and properties formalising the *displayability* domain.

$$
\begin{array}{l}
isIOOntologyWD: \\
\quad (\mathbf{isIOOntologyWD}(o) \Leftrightarrow \\
\quad isWDOntology(o) \wedge \{visible, hidden, critical, safe\} \subseteq instances(o) \wedge \\
\quad \{hasVisibility, hasCriticality\} = properties(o) \\
\quad \{visibility, criticality\} \subseteq classes(o)) \\
isVisibleWD: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\qquad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge i \in IOInstances \wedge \\
\qquad isIOOntologyWD(o) \Rightarrow (\mathbf{isVisibleWDi}(o, ipvs, i) \Leftrightarrow \\
\qquad\qquad CkeckOfSubsetOntologyInstances(o, ipvs) \wedge i \in dom(dom(ipvs))) \\
isVisiblei: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge \\
\quad i \in IOInstances \wedge isIOOntologyWD(o) \wedge isVisibleWDi(o, ipvs, i) \Rightarrow \\
\qquad (\mathbf{isVisiblei}(o, ipvs, i) \Leftrightarrow \\
\qquad instanceHasPropertyValuei(o, ipvs, i, hasVisibility, visible) \wedge \\
\qquad \neg instanceHasPropertyValuei(o, ipvs, i, hasVisibility, hidden)) \\
isHiddenWD: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge \\
\quad i \in IOInstances \wedge isIOOntologyWD(o) \Rightarrow \\
\qquad (\mathbf{isHiddenWD}(o, ipvs, i) \Leftrightarrow \\
\qquad\qquad CkeckOfSubsetOntologyInstances(o, ipvs) \wedge i \in dom(dom(ipvs))) \\
isHidden: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge \\
\quad i \in IOInstances \wedge isIOOntologyWD(o) \wedge isHiddenWD(o, ipvs, i) \Rightarrow \\
\qquad (\mathbf{isHidden}(o, ipvs, i) \Leftrightarrow \\
\qquad instanceHasPropertyValuei(o, ipvs, i, hasVisibility, hidden) \wedge \\
\qquad \neg instanceHasPropertyValuei(o, ipvs, i, hasVisibility, visible)) \\
isSafeWD: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge \\
\quad i \in IOInstances \wedge isIOOntologyWD(o) \Rightarrow \\
\qquad (\mathbf{isSafeWD}(o, ipvs, i) \Leftrightarrow \\
\qquad\qquad CkeckOfSubsetOntologyInstances(o, ipvs) \wedge i \in dom(dom(ipvs))) \\
isWDCriticali: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge \\
\quad i \in IOInstances \wedge isIOOntologyWD(o) \wedge isCriticalWD(o, ipvs, i) \Rightarrow \\
\quad (\mathbf{isCritical}(o, ipvs, i) \Leftrightarrow \\
\qquad instanceHasPropertyValuei(o, ipvs, i, hasCriticality, critical) \wedge \\
\qquad \neg instanceHasPropertyValuei(o, ipvs, i, hasCriticality, safe)) \\
isCriticali: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \wedge \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \wedge \\
\quad i \in IOInstances \wedge isIOOntologyWD(o) \wedge isCriticalWD(o, ipvs, i) \Rightarrow \\
\quad (\mathbf{isCritical}(o, ipvs, i) \Leftrightarrow \\
\qquad instanceHasPropertyValuei(o, ipvs, i, hasCriticality, critical) \wedge \\
\qquad \neg instanceHasPropertyValuei(o, ipvs, i, hasCriticality, safe))
\end{array}
$$

Listings 6.5: `DisplayabilityTheory` - axioms for tester operators

In addition, the predicate operators checking the content of an ontology and relationships between instances are axiomatised as well. For example, `isVisiblei` checks whether the interactive object is visible according to the ontology, i.e. if it is related to `visible` via the property `hasVisibility`. The predicate checks as well that the interactive object is not related to `hidden` for provability reasons. In the same vein, the other operators are axioma-

tised; see `isHiddeni`, `isSafei`, `isCriticali` in Listing 6.5. The operator `instanceHasPropertyValuei` of `OntologiesTheory` (see Listing 5.10) is always used in `DisplayabilityTheory` to check safety if an instance is associated with a specific value via a given property accordingly to the encapsulation principle. Similarly, all other operators are formalised and are associated with well-definedness conditions accordingly to the methodological rule previously stated. It is noteworthy that there exist shared terms in the well-definedness conditions of the predicate operators. They are those requiring that the ontology shall be a well-defined interactive object ontology (`isWDIOOntology`), the system state variable (`ipvs`) shall be a part of the ontology instance associations (`ckeckOfSubsetOntologyInstances`), and another common condition of all the well-definedness operators state that the interactive object `i` must belong to the model variable which is formalised as `i ∈ dom(dom(ipvs))`.

$$
\begin{array}{l}
isWDSetHiddeni: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad i \in IOInstances \land isIOOntologyWD(o) \Rightarrow \\
\qquad (\mathbf{isWDSetHiddeni}(o, ipvs, i) \Leftrightarrow \\
\qquad CkeckOfSubsetOntologyInstances(o, ipvs) \land i \in dom(dom(ipvs)) \land \\
\qquad isVisiblei(o, ipvs, i) \land (\forall j \cdot j \mapsto hasCriticality \mapsto critical \in ipvs \land \\
\qquad j \mapsto hasVisibility \mapsto visible \in ipvs)) \\
setHiddeni: \\
\quad \forall o, ipvs1, ipvs2, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land \\
\quad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad ipvs2 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad ipvs2 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad (ipvs2 = \mathbf{setHiddeni}(o, ipvs1, i) \Leftrightarrow \\
\qquad ipvs2 = (ipvs1 \cup i \mapsto hasVisibility \mapsto hidden) \setminus i \mapsto hasVisibility \mapsto visible) \\
isWDSetCriticali: \\
\quad \forall o, ipvs, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land \\
\quad ipvs \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad i \in IOInstances \land isIOOntologyWD(o) \Rightarrow \\
\quad\; (\mathbf{isWDSetCriticali}(o, ipvs, i) \Leftrightarrow \\
\qquad CkeckOfSubsetOntologyInstances(o, ipvs) \land isSafe(o, ipvs, i) \land i \in dom(dom(ipvs)) \land \\
\qquad \neg isHidden(o, ipvs, i) \land isVisiblei(o, ipvs, i) \land \\
\qquad (\forall j \cdot j \mapsto hasCriticality \mapsto critical \in ipvs \Rightarrow \\
\qquad\quad j \mapsto hasVisibility \mapsto visible \in ipvs)) \\
setCriticali: \\
\quad \forall o, ipvs1, ipvs2, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land \\
\quad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land \\
\quad i \in IOInstances \land isIOOntologyWD(o) \land isWDSetCriticali(o, ipvs1, i) \Rightarrow \\
\quad (ipvs2 = \mathbf{setCriticali}(o, ipvs1, i) \Leftrightarrow \\
\qquad ipvs2 = (ipvs1 \cup i \mapsto hasCriticality \mapsto critical) \setminus i \mapsto hasCriticality \mapsto safe)
\end{array}
$$

Listings 6.6: `DisplayabilityTheory` - axioms for accessor operators

The second class of operators provided by the `displayability` ontology are expression operators. They allow us to modify the ontology; specifically the instance associations representing the model state variable. In Listing 6.6, the `setCriticali` operator allows setting the `hasCriticality` property of an interactive object instance to `critical` value in a correct way. The definition of this operator states that when used within the scope of its well-definedness, it addresses one important domain safety property: *critical interactive objects must always be visible*. Thus, when the `hasCriticality` property of an interactive object is set to `critical`, its `hasVisibility` property is set to `visible` in accor-

dance with the domain safety properties. The well-definedness condition of the operator `setCriticali` is invoked by another defined operator `isCriticalWD` to ensure the correct use of the operator. In the same vein, the `setHiddeni` requires that the interactive object to be hidden shall not be critical; this domain safety rule is included in its well-definedness condition `isWDSetHiddeni`.

Theorems Definition

The last clause of the *Displayability* theory is `THEOREMS`; it is a key section of the theory since it contains safety domain-specific properties formulated and proved for all the operators intended to be used by the models. Listing 6.7 shows a set of theorems asserting that the operators must entail the required domain-specific safety properties, such as *critical objects must always be visible* which will be referred to as the *displayability* property. Therefore each operator, provided that it is well-defined, necessarily preserves this safety property. For example, the theorem `setCriticalThm` states that under the hypotheses that the arguments are well-typed and the well-definedness conditions of each operator hold, all objects with the property `critical` also have the property `visible` in the given ontology. In the same vein, `setVisibleThm`, `setHiddenThm`, and `setSafeThm` assert this *displayability* property is preserved by `setVisible`, `setHidden`, and `setSafe`.

$$
\begin{array}{l}
setCriticaliThm:\\
\quad \forall o, ipvs1, ipvs2, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land\\
\qquad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad ipvs2 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad i \in IOInstances \land isIOOntologyWD(o) \land isWDSetCriticali(o, ipvs1, i) \Rightarrow\\
\qquad (ipvs2 = \mathbf{setCriticali}(o, ipvs1, i) \Rightarrow\\
\qquad\; (\forall j \cdot j \mapsto hasCriticality \mapsto critical \in ipvs2 \Rightarrow j \mapsto hasVisibility \mapsto visible \in ipvs2))\\
setSafeiThm:\\
\quad \forall o, ipvs1, ipvs2, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land\\
\qquad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad ipvs2 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad i \in IOInstances \land isIOOntologyWD(o) \land isWDSetSafei(o, ipvs1, i) \Rightarrow\\
\qquad (ipvs2 = \mathbf{setSafei}(o, ipvs1, i) \Rightarrow\\
\qquad\; (\forall j \cdot j \mapsto hasCriticality \mapsto critical \in ipvs2 \Rightarrow j \mapsto hasVisibility \mapsto visible \in ipvs2))\\
setHiddeniThm:\\
\quad \forall o, ipvs1, ipvs2, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land\\
\qquad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad ipvs2 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad i \in IOInstances \land isIOOntologyWD(o) \land isWDSetHiddeni(o, ipvs1, i) \Rightarrow\\
\qquad (ipvs2 = \mathbf{setHiddeni}(o, ipvs1, i) \Rightarrow\\
\qquad\; (\forall j \cdot j \mapsto hasCriticality \mapsto critical \in ipvs2 \Rightarrow j \mapsto hasVisibility \mapsto visible \in ipvs2))\\
setVisibleiThm:\\
\quad \forall o, ipvs1, ipvs2, i \cdot o \in Ontology(IOClasses, IOProperties, IOInstances) \land\\
\qquad ipvs1 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad ipvs2 \in \mathbb{P}(IOInstances \times IOProperties \times IOInstances) \land\\
\qquad i \in IOInstances \land isIOOntologyWD(o) \land isWDSetVisiblei(o, ipvs1, i) \Rightarrow\\
\qquad (ipvs2 = \mathbf{setVisiblei}(o, ipvs1, i) \Rightarrow\\
\qquad\; (\forall j \cdot j \mapsto hasCriticality \mapsto critical \in ipvs2 \Rightarrow j \mapsto hasVisibility \mapsto visible \in ipvs2))
\end{array}
$$

Listings 6.7: `DisplayabilityTheory` - theorems

6.2.2 Instantiation of the Displayability Theory

The ontology of interactive objects, described in subsection 6.2.1, is instantiated to specifically formalise concepts and safety properties related to the interactive critical system of interest, namely the Traffic Collision Avoidance System (see subsection 4.1). The interactive objects of the TCAS are the graphical representation of aircraft.

```
CONTEXT
    InstantiationContext
CONSTANTS
    aircraftClass
    aircraftInstances
    aircraftOntology
    ClassProperties
    ClassInstances
    ClassAssociations
    instanceAssociation
    instanceAssociation0
    thingClass
    thingInstances
AXIOMS
    axm1 : partition(IOClasses, {thingClass}, {aircraftClass}, {visibility}, {criticality})
    axm2 : partition(IOInstances, aircraftInstances, {visible}, {hidden}, {safe}, {critical})
    axm3 : thingInstances = IOInstances
    axm4 : ClassProperties ∈ ℙ(IOClasses × IOProperties)
    axm5 : ClassProperties = {aircraftClass} × {hasVisibility, hasCriticality}
    axm6 : ClassInstances = ({aircraftClass} × aircraftInstances)∪
                    ({visibility} × {visible, hidden})∪
                    ({criticality} × {critical, safe})∪
                    ({thingClass} × thingInstances)
    axm7 : ClassAssociations ∈ ℙ(IOClasses × IOProperties × IOClasses)
    axm8 : ClassAssociations = ({aircraftClass} × {hasVisibility} × {visibility})∪
                  ({aircraftClass} × {hasCriticality} × {criticality})
    axm9 : instanceAssociation ∈ ℙ(IOInstances × IOProperties × IOInstances)
    axm10 : instanceAssociation = (aircraftInstances × {hasVisibility} × {hidden, visible})∪
                    (aircraftInstances × {hasCriticality} × {safe, critical})
    axm11 : instanceAssociation0 ∈ ℙ(IOInstances × IOProperties × IOInstances)
    axm12 : instanceAssociation0 = (aircraftInstances × {hasVisibility} × {hidden})∪
                    (aircraftInstances × {hasCriticality} × {safe})
    instAssoc0Thm : instanceAssociation0 ⊆ instanceAssociation
    arcrftOntoDef : aircraftOntology = consOntology(IOClasses, IOProperties, IOInstances,
                    ClassProperties, ClassInstances, ClassAssociations, instanceAssociation)
    ConformThm : isIOOntologyWD(aircraftOntology)
    isAThm : isA(aircraftOntology, aircraftClass, thingClass)
END
```

Listings 6.8: Context for displayability theory instatiation

Listing 6.8 presents the Event-B context obtained by instantiating the domain theory `DisplayabilityTheory`. In this Event-B context, the axioms (`axm1-axm14`) define or extend the seven components required for the complete instantiation of the ontology data type (see subsection 5.2.2). `IOClasses` and `IOInstances` are instantiated as enumerated sets in `axm1` and `axm2`, respectively. `aicraftClass` and `aircraftInstances` (containing all the classes and instances of the aircraft interactive object ontology, respectively) are introduced as constants. Also, a set of axioms (`axm3-axm10`) are used to extend the definition of the ontology components by providing the necessary class properties and class associations information. In particular, `thingClass` and `thingInstances` are defined which represent the root of all ontology classes and the set of all ontol-

ogy's instances, respectively. ClassProperties is extended by the properties of the class aircraftClass. classInstances is also defined where aircrafClass, Visibility and Criticality classes are associated with their instances. Also, instanceAssociations is defined to contain the set of triples (instance, property and value) which represent the assertional part of the domain knowledge.

Finally, the ontology aircraftOntology is built in aircraftOntologyDef axiom using the components previously defined using the ontology constructor consOntology by setting the seven parameters IOClasses, IOProperties, IOInstances, ClassProperties, ClassInstances, instanceAssociation and ClassAssociations.

An important verification, which shall always be performed at the instantiation phase, is verifying that the ontology is well-defined. In Listing 6.8, ConformThm fills this requirement, that is the ontology components are correctly defined in the sense formalised by the isWDIOOntology predicate operator (see Listing 6.5). It is defined using isWDOntology predicate operator which checks four conditions: (1) that the classes have authorised properties and (2) instances, (3) that classes associations are correct meaning that classes are related to classes using properties associated with the source class (isWDClassAssociation), and (4) that instance associations are valid according to class associations (isWDClassInstances). Additionally, isWDIOOntology checks more conditions which are related to interactive objects domain as the ontology contains visible, critical instances, and properties as hasVisible. Last, the theorem isAThm states that aircrafClass is a subclass of thingClass which is proved straightforwardly since the definition of this operator relies on the inclusion of instances. This theorem is defined to showcase other statement that may be checked on ontologies.

In the following, subsection 6.2.3 illustrates the classical modelling way where the properties of the systems are specified and proved as invariants of the model. Then, subsection 6.2.4 demonstrates the advocated way of modelling (i.e. with an *explicit* representation of the domain) where the properties are *transferred* from the domain theory to the model. Indeed, the visibility properties, stating that critical aircraft are always visible (see **REQ10** in section 4.1), is specified and proved as a theorem of the model.

6.2.3 Modelling without the Theory Operators

This subsection is dedicated to the first development of the TCAS model without using the annotation mechanism provided by the domain knowledge theory (displayability ontology) as well as the operators and their well-definedness predicates. This modelling paradigm is close to the classical paradigm in Event-B showcased in subsection 5.1.2 where modifications on the state variable are realised with the built-in Event-B operations in contrast to using domain-specific predefined primitives.

The TCAS user interface model is formalised as an Event-B machine (see Listing 6.9), where the state of the model is represented by a single Event-B variable: system which is part of the ontology of displayability. It represents

the visibility state of an aircraft close to the host aircraft (visible or hidden), and their level of criticality (safe or critical). In Listing 6.9, the `typing` invariant gives the Event-B type of the `system` state variable. It is a subset of `instanceAssociation`(of type $\mathbb{P}$(`IOInstances`$\times$`IOProperties`$\times$`IOInstances`)). In this model, safety property `safetyInv` of the TCAS, stating that *critical aircraft shall be visible* is defined.

```
MACHINE
  SetTheoriticOperationsBasedModel
SEES
  InstantiationContext
VARIABLES  system
INVARIANTS
  typing : system ⊆ instanceAssociation
  safetyInv : ∀io · io ∈ dom(dom(system))
                ⇒ (io ↦ hasCriticality ↦ critical ∈ system ⇒
                              io ↦ hasVisibility ↦ visible ∈ system)
```

Listings 6.9: TCAS model based on set operations - invariants

```
EVENTS
  INITIALISATION
  THEN
    act1 : system := instanceAssociation0
  END
  makeAircraftCritical
  ANY  i , safeEntry , crticalEntry , hiddenEntry , visibleEntry
  WHERE
    grd1 : i ∈ dom(dom(system))
    grd2 : safeEntry = {i ↦ hasCriticality ↦ safe}
    grd3 : safeEntry ⊆ system
    grd4 : crticalEntry = {i ↦ hasCriticality ↦ critical}
    grd5 : crticalEntry ⊈ system
    grd6 : hiddenEntry = {i ↦ hasVisibility ↦ hidden}
    grd7 : visibleEntry = {i ↦ hasVisibility ↦ visible}
  THEN
    act1 : system := (((system \ safeEntry) ∪ crticalEntry) \ hiddenEntry) ∪ visibleEntry
  END
  makeAircraftVisible  . . .
  makeAircraftSafe  . . .
  makeAAircraftHidden
  ANY  i , safeEntry , crticalEntry , hiddenEntry , visibleEntry
  WHERE
    grd1 : i ∈ dom(dom(system))
    grd2 : safeEntry ∈ ℙ(IOInstances × IOProperties × IOInstances)
    grd3 : safeEntry = {i ↦ hasCriticality ↦ safe}
    grd4 : crticalEntry ∈ ℙ(IOInstances × IOProperties × IOInstances)
    grd5 : crticalEntry = {i ↦ hasCriticality ↦ critical}
    grd6 : hiddenEntry ∈ ℙ(IOInstances × IOProperties × IOInstances)
    grd7 : hiddenEntry = {i ↦ hasVisibility ↦ hidden}
    grd8 : hiddenEntry ⊈ system
    grd9 : visibleEntry ∈ ℙ(IOInstances × IOProperties × IOInstances)
    grd10 : visibleEntry = {i ↦ hasVisibility ↦ visible}
    notCritical : crticalEntry ⊈ system ∧ safeEntry ⊆ system
  THEN
    act1 : system := (system \ visibleEntry) ∪ hiddenEntry
  END
END
```

Listings 6.10: TCAS model based on set operations - events

In addition, a collection of events is defined to model the progress of the model. In Listings 6.10, an event `makeAircraftCritical` formalises the up-

dating of `system` variable where some aircraft becomes critical, making it visible to preserve the invariant `safetyInv`. Note that the definition of event's action is deduced from the understanding of the requirements of the TCAS (see section 4.1), and so does the formalisation of the guards. For example, in `MakeAircraftHidden`, **notCritical** guard stipulates that the aircraft being hidden must not be critical. Again this information is derived from the understanding of the TCAS requirements. Besides, two other events are defined for updating some aircraft states: `makeAircraftVisible` and `makeAircraftSafe`. `INITIALISATION` event uses a constant defined in `InstantiationContext` (see Listing 6.8). At the beginning, all aircraft are considered hidden and safe. The invariants are proven inductively where every event shall preserve the safety property. Furthermore, the correct guards of each event are designed from scratch instead of being derived systematically from the definition of the actions.

Observation. The developer needs to formalise all the domain properties since the operators are not used. Indeed, the actions and the guards of the events are derived from the requirements so every model may interpret it differently. Moreover, the proof must be performed inductively. Such a process is error-prone and has to be repeated for every model.

The goal of the improved modelling (see the next section) is to remedy these downsides. Specifically, the definitions of the events shall be built on theory primitives so they may benefit from the safety properties they entail.

6.2.4 Modelling with the Theory Operators

This second version of the modelling of the TCAS uses the framework presented in section 6.1. The modelling paradigm is the same as that showcased in the temperature aggregator didactic example in subsection 5.1.3.

MACHINE $TheoryOperatorsBasedModel$
SEES $InstantiationContext$
VARIABLES $system$
INVARIANTS
  typing : $CkeckOfSubsetOntologyInstances(aircraftOntology, system)$
  UsedOps : $\exists ipvs, i\cdot$
$(isWDSetSafei(aircraftOntology, ipvs, i)\wedge$
$system = setSafei(aircraftOntology, ipvs, i))\vee$
$(isWDSetCriticali(aircraftOntology, ipvs, i)\wedge$
$system = setCriticali(aircraftOntology, ipvs, i))\vee$
$(isWDSetVisiblei(aircraftOntology, ipvs, i)\wedge$
$system = setVisiblei(aircraftOntology, ipvs, i))\vee$
$(isWDSetHiddeni(aircraftOntology, ipvs, i)\wedge$
$system = setHiddeni(aircraftOntology, ipvs, i))$
  safetyThm : $\forall i\cdot i\mapsto hasCriticality\mapsto critical\in system\Rightarrow$
$i\mapsto hasCriticality\mapsto safe\in system$

Listings 6.11: TCAS model based on theory operators - invariants

The domain knowledge constraints and concepts are transferred by annotating the `system` state variable. In Listing 6.11, annotation is achieved by the predicate operator `CkeckOfSubsetOntologyInstances` which checks that the

state variable remains in the allowed instance associations of the specified ontology `aircraftOntology`. The model has one variable `system` which is a subset of `instanceAssociation` typed as $\mathbb{P}$(`IOInstances`$\times$`IOProperties`$\times$`IOInstances`).

Invariant `usedOps` is introduced to formalise the methodological rule of using only the operators provided by the ontology of interactive objects. The main consequence of complying with this guideline is to inherit the theorems proved in the theory once and for all. In particular, the safety property states that *critical aircraft shall be visible* (**REQ10** safety requirement; see section 4.1). This safety property is formalised as a theorem `safetyThm` of the model instead of being proved inductively.

A key difference compared to the first version of the TCAS modelling is the use of operators and well-definedness conditions. In this model, the events are defined using the operators of the displayability ontology theory. Event `makeAircraftCritical` uses the operator `setCriticali` to correctly update the status of an aircraft becoming close enough to own aircraft to be considered critical. Furthermore, the well-definedness condition of this operator must be fulfilled to be applied safely. Therefore, `isWDSetCriticali` is added as a guard to the event. It is noteworthy that the process of determining the conditions of the correct application of the operators is straightforward since the well-definedness condition are already available in the theory of displayability.

```
EVENTS
  INITIALISATION
  THEN
    act1 : system := instanceAssociation0
  END
  makeAircraftVisible ...
  makeAircraftSafe ...
  makeAircraftCritical
  ANY i
  WHERE
    grd1 : i ∈ dom(dom(system))
    grd2 : ontologyContainsInstances(aircraftOntology, {i})
    grd3 : isVisibleWDi(aircraftOntology, system, i)
    grd4 : isVisiblei(aircraftOntology, system, i)
    grd5 : isSafeWD(aircraftOntology, system, i)
    grd6 : isSafe(aircraftOntology, system, i)
    grd7 : isWDSetCriticali(aircraftOntology, system, i)
  THEN
    act1 : system := setCriticali(aircraftOntology, system, i)
  END
  makeAAircraftHidden
  ANY i
  WHERE
    grd1 : i ∈ dom(dom(system))
    grd2 : ontologyContainsInstances(aircraftOntology, {i})
    grd3 : isWDSetHiddeni(aircraftOntology, system, i)
  THEN
    act1 : system := setHiddeni(aircraftOntology, system, i)
  END
END
```

Listings 6.12: TCAS model based on theory operators - events

Since this second version modelling of TCAS follows the explicit domain knowledge modelling, domain safety properties are not proved inductively, meaning that there is no need to perform proof for each event that the safety prop-

erties are preserved: they are specified as theorems and proved deductively and straightforwardly, i.e., these safety properties follow logically from the theory operators, their well-definedness and the theorems of the displayability theory. Additionally, trivial typing invariant and the unique working hypothesis, which requires that *only the theory operators allowed to be used*, need to be discharged inductively, and their proofs are direct since each event only uses the operators provided by the theory. For example, event `makeAAircraftHidden` uses, in its action clause, operator `setHiddeni` whose well-definedness `isWDSetHidde ni` predicate is included in the guards. In addition, theorems formalising domain properties are discharged using the *modus ponens* inference rule. The theorems defined in the theory and the working hypothesis are sufficient to deduce the domain properties.

6.2.5 Proof Statistics

Event-B Component	**Automatic**	**Interactive**	**Total POs**
Temperature aggregator example (chapter 5)			
`TemperatureTheory`	4 (29%)	10 (71%)	14 (100%)
`C_TemperatureContext`	0	0	0
`C_TempertureMachine`	1 (20%)	4 (80%)	5 (100%)
`T_TempertureMachine`	**6 (60%)**	**4 (40%)**	**10 (100%)**
TCAS case study			
`OntologiesTheory`	0 (0%)	21 (100%)	21 (100%)
`DisplayabilityTheory`	0 (0%)	16 (100%)	16 (100%)
`InstantiationContext`	1 (25%)	3 (75%)	4 (100%)
`SetTheoriticOperationsModel`	2 (100%)	8 (80%)	10 (100%)
`TheoryOperatorsModel`	**10 (45%)**	**12 (55%)**	**22 (100%)**

Table 6.1: Proof statistics of OML and TCAS case study

The development of two versions of each case study (temperature aggregator in chapter 5 and TCAS study in this chapter 6) demonstrates reduction in proof effort (see Table 6.1). Indeed, proving time decreases on the long run since the theories are developed and proved once and for all, and proof of theorems of the theory needed in the model development is straightforward. The proofs are significantly streamlined and become repetitive therefore amenable to automation by writing inference and rewrite rule in the Event-B theory. In practice, the totality of the proof obligations is divided into three categories. The first category contains the POs generated for ensuring the well-definedness of the actions of the events using the operators, and they are discharged trivially during the development as the well-definedness operators are included in the guards. The second category represents the condition of exclusively using the theory operators, they are also discharged trivially when the working hypothesis, requiring that only the data types and operators of the theory are used in system modelling, is followed. The last category is related to theorems formalising the safety properties, and they are also discharged straightforwardly

thanks to the universally quantified theorems of the theory —using the *modus ponens* inference rule.

Table 6.1 sums up the numbers of POs generated for each development. The first observation is that the number of POs is greater when using theory data types and operators. The second observation is that a bigger part of the POs are not discharged automatically in the case when the models are built on theory primitives. The first issue is fundamentally intrinsic to the methodology and, is grouped in the second category of POs. The second inconvenient is technical and, may be overcome with the help of providing a collection of proof rules. Fortunately, the Rodin proving infrastructure provides mechanisms to write and integrate such rules: inference and rewrite rules.

In conclusion, the models developed using the methodology of section 6.1 require less proving effort than those using the `OntologiesTheory` Event-B theory and `DisplayabilityTheory` Event-B theory . In addition, the investment of developing the generic theory is more profitable on the long run. Indeed, the developed theory, including operators and theorems, is reusable by other models instead of having to be defined everything from scratch.

6.3 Conclusion

This chapter proposed a framework for the annotation of variables of a system's formal model entailing the transfer of the predefined domain-specific safety properties. The TCAS case study (see section 4.1) has been addressed following the two ways of modelling. First, classical modelling of the TCAS where safety property as an invariant had been developed. In this case, the safety property had to be proved inductively for every event. Second, the annotation-based modelling of the TCAS had been achieved using an Event-B theory of displayability. In this second case, the operators of the domain theory have been used so that the safety property were specified as a theorem of the model.

The approach advocated, based on an explicit modelling of the domain knowledge and annotation of the model, has the benefit of reducing the modelling effort since it exempts the designer from formalising domain properties in the model. Moreover, these common domain-specific properties are described once and for all, and they are transferred from the theory thanks to the methodological rule stipulating the exclusive use of the operators. Furthermore, the necessary condition for applying the operators are automatically determined since they are available in the well-definedness condition clause of each operator. Moreover, the framework has an important structural advantage; the models are described at a higher level of abstraction since it relies on domain theory primitives rather than on boilerplate set-theoretic code. From a property specification point of view, the framework ensures that if the system model is built using the domain ontology, it necessarily implies all the theorems formalising the domain rules. Therefore, the framework allowed for a reduction of the proving effort on the long run.

Chapter 7

Annotation-Based Analysis of Behavioural Properties

This Chapter contains:

7.1 Our Approach 92
7.2 The Event-B Meta-Theory 93
7.2.1 Event-B Machine Structure as a Data Type 93
7.2.2 Event-B Machine Proof Obligations as Predicates 95
7.2.3 Modelling with Event-B Meta-Theory 96
7.3 A Framework for Behavioural Analyses 96
7.3.1 The Architecture of the Framework 97
7.3.2 How does the Framework Work? 98
7.4 The Framework at Work 100
7.4.1 Defining a Domain-Specific Behavioural Analysis 100
7.4.2 Applying a Domain-Specific Behavioural Analysis 106
7.5 Advantages of the Framework 111
7.5.1 Principled Approach and Reusability 111
7.5.2 Non-intrusiveness 111
7.5.3 Verification Based on Theorem Proving 111
7.5.4 Proof and Modelling Effort Reduction 112
7.5.5 Generalisation 113
7.6 Conclusion 113

This chapter presents the formal framework designed to allow behavioural analyses to refer to domain knowledge models, thereby checking the domain knowledge liveness properties when designing formal models. Moreover, the proposed approach is investigated and illustrated in the real-world example of an Automatic Machine Teller (ATM) user interface. The approach is based on annotation of design models: the events are typed with domain knowledge concepts and manipulated with custom operators yielding the properties to be enforced on the models. The ontology modelling language allows the definition

of domain knowledge concepts and constraints which are then checked on the design model through the annotation mechanism.

The chapter is organised as follows. Section 7.1 provides an overview of the framework for expressing domain-specific behavioural analyses and introduces its key elements and building blocks. Section 7.2 presents an essential building block of the framework: reflexive Event-B. It is a formalisation of Event-B concepts using Event-B in the form of a generic Event-B theory. The framework relies on the work presented in [143] which describes the reflexive Event-B framework to manipulate and annotate Event-B components. Section 7.3 discusses the framework for defining non-intrusive domain-specific behavioural analyses in Event-B. Section 7.4 showcases the framework on a real-world case study: automatic teller machine. Last, section 7.5 enumerates a number of benefits of using the framework for defining and applying domain-specific behavioural analyses.

7.1 Our Approach

The behavioural analysis of state-based models may be enhanced by considering domain knowledge constraints when investigating the models. To enable this kind of analysis, a solution lies in describing domain knowledge constraints, especially domain knowledge related to behaviours. This domain knowledge is often specified in domain requirements or standards documents. This formal ontology of behaviours may then be used to annotate actions and events of formal models in a non-intrusive manner in the sense that the analysis may be applied at any level of the refinement chain of system modelling. Next, behavioural analyses using domain constraints may be defined and applied to formal models.

Figure 7.1 depicts the general view of the framework proposed for addressing the problem of defining and applying domain-specific behavioural analyses. It is composed of two basic blocks: ontology modelling language (see Section 5.2) and reflexive Event-B framework (see Section 7.2). The first component provides primitives to write domain concepts and constraints as ontologies, while the second component allows for the abstraction of a system as an instance of the Event-B meta-theory language. The meta-level paves the way to reasoning extension specifically providing a language for expressing user-defined proof obligations. Moreover, all the behavioural analyses encoded in first-order logic can be expressed and validated on models in this framework. In addition to temporal properties, the framework allows the expression of enriched analyses, like checking that a class of events is eventually followed by another class of events, that take domain knowledge into account. A mechanism for referencing domain knowledge in design models is also defined through annotation.

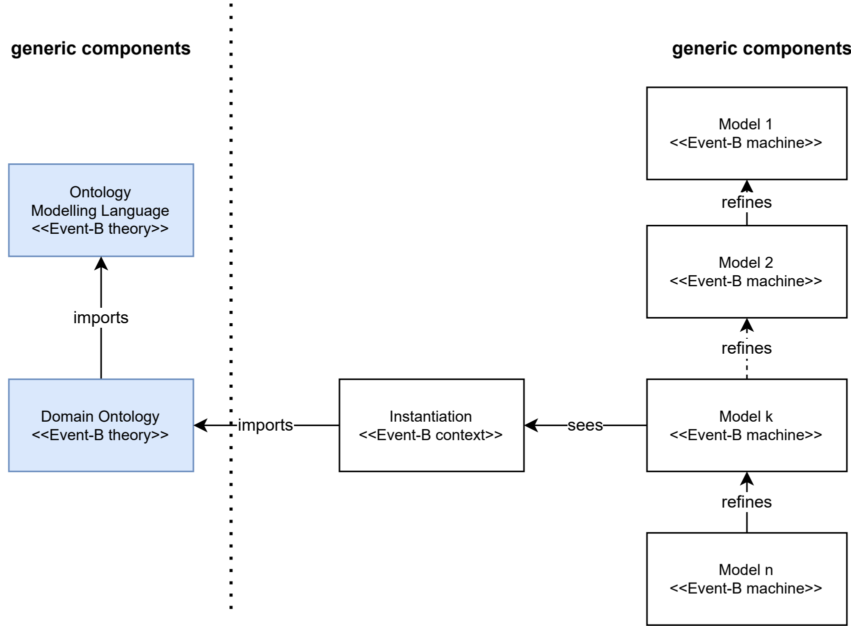


Figure 7.1: A framework for domain-specific behavioural analyses

7.2 The Event-B Meta-Theory

In [143], a reflexive framework has been defined for enabling the extension of the reasoning capabilities of Event-B. This EB4EB framework is an Event-B-based modelling framework allowing to manipulate Event-B features explicitly based on meta-modelling concepts. This framework relies on a set of Event-B theories defining data types, operators with well-definedness conditions, theorems and proof rules. It preserves the core logical foundation, and semantics, of classical Event-B. It is a core block of the framework highlighted in Figure 7.1 where it allows us to express analyses applicable to Event-B models since the concepts of Event-B language are available as first-class elements. Therefore, this section overviews the structure of Event-B meta-theory[1].

7.2.1 Event-B Machine Structure as a Data Type

The first section of the theory formalising Even-B concepts consists of data types. Listing 7.1 shows the data type representing the machine's elements, which are parameterised by two types: `Ev` (i.e. transitions) and `St` (i.e. states). A constructor is defined (`Cons_machine`) where each argument corresponds to a machine component. The machine denotes a state transition system constrained by the invariant (`Inv`).

[1] The full listing of `EvtBTheo` is in appendix A.2

Events are triggered by the initialisation event (`Init`) and then by progress events (`Progress`). State changes are provoked by the After predicate (`AP`) for `Init`, and the Before After Predicate (`BAP`) for progress events if their corresponding guards (`Grd`) are true. A numeric variant (`Variant`) is defined, and it is used for liveness properties. Moreover, `Ordinary`, `Anticipated`, and `Convergent` define a partition of events for the instantiated machine.

```
THEORY EvtBTheo
  TYPE PARAMETERS STATE, EVENT
  DATA TYPES Machine(STATE, EVENT)
    CONSTRUCTORS
      Cons\_machine(
        Event : ℙ(EVENT),
        State : ℙ(STATE),
        Init : EVENT,
        Progress : ℙ(EVENT),
        Variant : ℙ(STATE × ℤ),
        Convergent : ℙ(EVENT),
        Anticipated : ℙ(EVENT),
        Ordinary : ℙ(EVENT),
        AP : ℙ(STATE),
        BAP : ℙ(EVENT × (STATE × STATE)),
        Grd : ℙ(EVENT × STATE),
        Inv : ℙ(STATE),
        Thm : ℙ(STATE))
```

Listings 7.1: `EvtBTheo` - data type

Well-Constructed machines. The data type requires formalising the constraints on the constructor's arguments specified in the Event-B book [3].

```
BAP_WellCons <predicate> (m : Machine(STATE, EVENT))
  direct definition
    dom(BAP(m)) = Progress(m)
Grd_WellCons <predicate> (m : Machine(STATE, EVENT))
  direct definition
    dom(Grd(m)) = Progress(m)
Event_WellCons <predicate> (m : Machine(STATE, EVENT))
  direct definition
    partition(Event(m), {Init(m)}, Progress(m))
Variant_WellCons <predicate> (m : Machine(STATE, EVENT))
  direct definition
    Inv(m) ◁ Variant(m) ∈ Inv(m) → ℤ
Tag_Event_WellCons <predicate> (m : Machine(EVENT, STATE))
  direct definition
    partition(Event(m), Ordinary(m), Convergent(m), Anticipated(m))∧
    Init(m) ∈ Ordinary(m)
Machine_WellCons <predicate> (m : Machine(STATE, EVENT))
  direct definition
    BAP_WellCons(m)∧
    Grd_WellCons(m)∧
    Event_WellCons(m)∧
    Tag_Event_WellCons(m)∧
    Variant_WellCons(m)
```

Listings 7.2: `EvtBTheo` - operators Event-B machine well-definedness

For example, `Event_WellCons` (see Listing 7.2) encodes the property stating that events are partitioned as initialisation event and progress events and `Machine_WellCons` defines well-constructed machines. This predicates operator checks five conditions on the data type attributes (1) `BAP_WellCons` checks

that only the progress events are associated with Before After Predicate excluding the initialisation event; (2) in the same vein, `Grd_WellCons` verifies that only progress events are associated with guards since, by definition, the initialisation event is not; (3) `Event_WellCons` checks that instantiation provides one initialisation event and all the other events belong to progress; (4) `Tag_EventWellCons` is a predicate operator that checks that the events are partitioned into ordinary, convergent, and anticipated accordingly to the specification in the Event-B book; (5) `Variant_WellCons` verifies the well-definition of the variant stipulating that the variant is a value associated with every state in the invariant state domain (i.e. it is a total function whose domain is the set of states representing the invariant).

7.2.2 Event-B Machine Proof Obligations as Predicates

```
Mch_THM <predicate> (m : Machine(STATE, EVENT))
   direct definition
      Inv(m) ⊆ Thm(m)
Mch_INV_Init <predicate> (m : Machine(STATE, EVENT))
   direct definition
      AP(m) ⊆ Inv(m)
Mch_INV_One_Ev <predicate> (m : Machine(STATE, EVENT) , e : EVENT)
   well−definedness e ∈ Progress(m)
   direct definition
      BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]] ⊆ Inv(m)
Mch_INV <predicate> (m : Machine(STATE, EVENT))
   direct definition
      Mch_INV_Init(m)∧
      (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
Mch_FIS_Init <predicate> (m : Machine(STATE, EVENT))
   direct definition
      Inv(m) ∩ AP(m) ≠ ∅
Mch_FIS_One_Ev <predicate> (m : Machine(STATE, EVENT) , e : EVENT)
   well−definedness e ∈ Progress(m)
   direct definition
      Inv(m) ∩ Grd(m)[{e}] ⊆ dom(BAP(m)[{e}])
Mch_FIS <predicate> (m : Machine(STATE, EVENT))
   direct definition
      Mch_FIS_Init(m)∧
      (∀e · e ∈ Progress(m) ⇒ Mch_FIS_One_Ev(m, e))
Mch_VARIANT_One_Ev <predicate> (m : Machine(STATE, EVENT) , e : EVENT ,
      s : STATE)
   well−definedness Variant_WellCons(m) , Mch_INV_One_Ev(m, e) , e ∈ Progress(m)
         , e ∈ Convergent(m) , s ∈ Inv(m) , s ∈ Grd(m)[{e}]
   direct definition
      ∀sp · sp ∈ BAP(m)[{e}][{s}] ⇒ (Inv(m) ◁ Variant(m))(s) > (Inv(m) ◁ Variant(m))(sp)
Mch_VARIANT <predicate> (m : Machine(STATE, EVENT))
   well−definedness Variant_WellCons(m) , Mch_INV(m) , BAP_WellCons(m) ,
         Tag_Event_WellCons(m) , Event_WellCons(m)
   direct definition
      ∀e, s · e ∈ Event(m) ∧ e ∈ Convergent(m) ∧ s ∈ State(m) ∧ s ∈ Inv(m)∧
            s ∈ Grd(m)[{e}] ⇒ Mch_VARIANT_One_Ev(m, e, s)
Mch_NAT_One_Ev <predicate> (m : Machine(STATE, EVENT) , e : EVENT)
   well−definedness e ∈ Convergent(m)
   direct definition
      Variant(m)[Inv(m) ∩ Grd(m)[{e}]] ⊆ ℕ
Mch_NAT <predicate> (m : Machine(STATE, EVENT))
   direct definition
      Variant(m)[Inv(m) ∩ Grd(m)[Convergent(m)]] ⊆ ℕ
END
```

Listings 7.3: `EvtBTheo` - operators for checking Event-B proof obligations

The proof obligations are formalised as follows. Each proof obligation is formalised using set theory. Predicates over state variables are modelled as sets of states satisfying the predicate and logical connectives are formalised by operations on sets (see Listing 7.3).

Listing 7.3 describes the induction principle for verifying the invariant PO where `Mch_INV_Init` predicate states that the initialisation event must establish the invariant ($AP(m) \subseteq Inv(m)$) and `Mch_INV_One_Ev` states that a given progress event e must preserve the invariant ($BAP(m)[\{e\}]\ [Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$). Last, the `Inv` PO is formalised by the `Mch_INV` operator as the conjunction of the two previous predicate operators. Likewise, all Event-B proof obligations are formalised using this translation from predicate logic to set theory: `Mch_FIS`, `Mch_VARIANT` and `Mch_NAT`.

Last, the operator `check_Machine_Consistency` (see Listing 7.4) is the key predicate for checking that a machine instance is correct in the sense of Event-B (all associated proof obligations are discharged). It is the conjunction of all the predicates formalising Event-B proof obligations.

```
check_Machine_Consistency <predicate> (m : Machine(STATE, EVENT))
  well−definedness Machine_WellCons(m)
  direct definition
    Mch_THM(m)∧
    Mch_INV(m)∧
    Mch_FIS(m)∧
    Mch_NAT(m)∧
    Mch_VARIANT(m)
```

Listings 7.4: `EvtBTheo` - operator for Event-B machine consistency

When this predicate is used as a `theorem` in an Event-B system development then the core PO as well as the well-definedness PO are automatically generated by the Rodin platform. Discharging all the generated PO along with this theorem ensures the consistency of the machine.

7.2.3 Modelling with Event-B Meta-Theory

Event-B meta-theory is instantiated to define specific Event-B machines. Instantiation consists in defining an Event-B context with instances for the type parameters `St` and `Ev` and providing instances for the attributes of `Cons_machine`. Event-B meta-theory is instantiated to model a particular system or to transform the existing Event-B model from classical Event-B to its meta-model. The instantiation consists in defining an Event-B context providing concrete carrier sets for the type parameters `St` and `Ev` of the `EvtBTheo` theory − which represents the packed Cartesian product of the types of machine variables and the set of the names of its events, respectively.

7.3 A Framework for Behavioural Analyses

The framework for defining and applying domain-specific behavioural analyses relies on two theories presented previously `OntologiesTheories` and `EvtBTheo`.

It allows the description of domain-specific behavioural properties as generic predicate operators expressed in terms of Event-B machine concepts.

Figure 7.2 illustrates the general architecture of the framework devised for defining and applying domain-specific behavioural analyses. It highlights the three distinct building blocks: domain-specific components (coloured in red), behavioural-related components (coloured in green) and the component defining and applying the analyses (coloured in purple).

7.3.1 The Architecture of the Framework

The framework is composed of three main parts: Event-B refinement-based development chain labelled with the letter (A) in Figure 7.2, Event-B theories representing the generic and reusable blocks of the framework which is labelled with a letter (B) in Figure 7.2 and instances representing the blocks which are specific to the system under study and therefore they need to be defined specifically for the system under study; they are described as Event-B contexts. This last part is labelled with the letter (C) in Figure 7.2.

Nota bene: The definition of domain-specific behavioural analysis requires two basic theories: (1) a theory formalising a language capable of expressing domain concepts and constraints, and (2) a theory formalising the meta-level of the modelling language. It is noteworthy that this framework could be replicated on other formal modelling languages (other than Event-B). However, this is true provided that the language supports state-based semantics and provides generic modules supporting algebraic specification (data types and operators with well-definedness predicates).

(A) Event-B Development

This part covers the Event-B refinement-based development of the system under study. Figure 7.2 highlights the fact the analysis is not intrusive in the sense that it does not alter the process of the modelling. Moreover, this framework may be used to analyse existing models as well thanks to the exporting step.

(B) Theories

This part is composed of three Event-B theories: `OntologiesTheory`, `EvtBTheo`, `BehaviouralPropertiesTheory`. The first and second theories are used by the third to integrate domain-specific constraints in behavioural analysis.

- `OntologiesTheory` (see section 5.2) provides the ontology modelling language for formalising the domain of action and events.

- `EvtBTheory` (see section 7.2) whose role is to provide a way of manipulating the Event-B concepts allowing the definition of custom proof obligations.

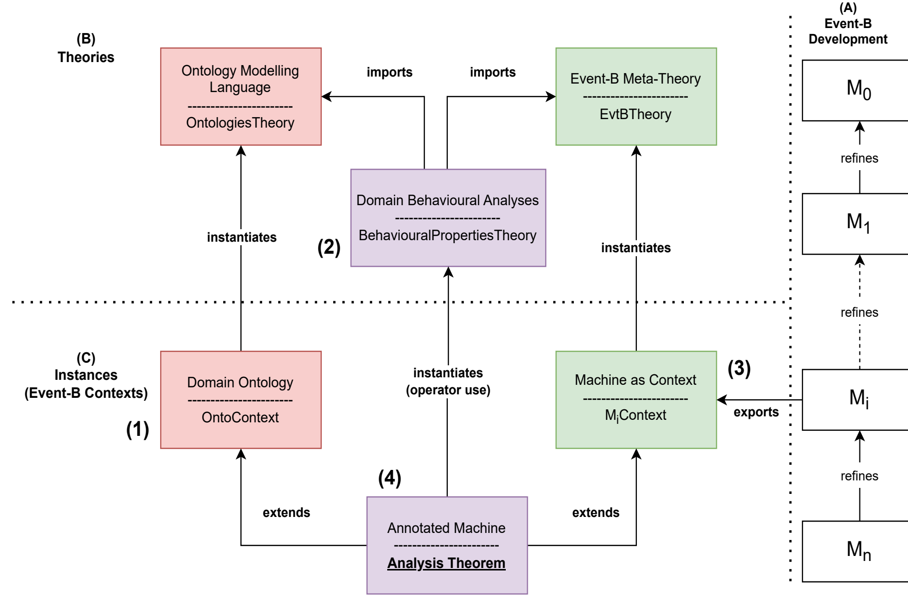


Figure 7.2: Fine-grain view of the behavioural analysis framework

- `BehaviouralPropertiesTheory` importing both theories for specifying new proof obligations as predicate operators expressing specific behavioural domain properties and used as theorems on specific Event-B models expressed as instances. The analysis is defined as a predicate operator parameterised by the domain and applied to the Event-B machine.

(C) Instances (Event-B Contexts)

The third part is composed of the contexts instantiating the theories of Part (B). Context (4) describes the annotated model, which extends the ontology context, and the Event-B model context, and defines theorems corresponding to the behavioural properties to be checked. The proof of these theorems, `Analysis Theorem` represented in step (4) on the Figure 7.2, guarantees that the properties hold on the analysed machine $\mathtt{M}_i$ of part (A).

7.3.2 How does the Framework Work?

The framework may be used by following numbered steps on Figure 7.2. These steps may be grouped into two main activities: the definition of the analysis and the application of the analysis. Steps (1) and (2) compose the definition of the analysis which is engineered once and then may be applied to analyse numerous systems. Steps (3) and (4) form the activity of applying the analysis to a specific system.

Step 1: Define an Ontology of Events

The first step of the framework consists in formalising the domain of events to handle the behaviours of systems. Typically, it involves specifying the domain requirements for actions and events as an ontology. Event-B contexts are used to contain this ontology, which is an instance of the theory of ontology modelling language presented in Section 5.2.

Step 2: Specify Domain-Specific Behavioural Analysis

Afterwards, the analysis is defined by specifying the behavioural analysis as parameterised by the event ontology. It consists in defining a predicate operator formalising the behavioural analysis. The analysis definition resembles the definition of custom proof obligations integrating domain-specific constraints. Functionally, the analysis is a parametric predicate operator requiring at least two key arguments: an ontology and an instance of the Event-B meta-theory to be analysed. This predicate operator may require other pieces of information on the system being investigated to efficiently achieve the analysis.

Step 3: Export The Event-B Machine

The second phase of using the framework consists in applying domain-specific behavioural analyses. The immediate step consists in exporting the Event-B model meaning converting it to an instance of Event-B meta-theory. The produced instance allows the explicit manipulation of the machine elements using operators of the theory (see Section 7.2). The translation is straightforward thanks to the one-to-one correspondence between the data type attributes declared in `EvtBTheo` and an Event-B machine parts.

Step 4: Annotate The Event-B meta-theory Instance

The last step of the application of the analysis is checking the analysis on a particular machine. The checking of the analysis is achieved by the predicate formalising the analysis as a theorem. This step requires annotating the Event-B meta-theory instance, resulting from the exporting, with the ontology concepts defined in Step 1. Machine concepts (variables and events) are linked to ontology concepts (i.e. being the tags in the annotation step). Finally, the annotated machine is analysed by operators of the domain behavioural theory (`BehaviouralPropertiesTheory`). Checking the analysis means proving the theorem stating that the analysis predicate operator may be deduced from the definitions and axioms of the Event-B model and domain ontology.

7.4 The Framework at Work

This section illustrates how the framework is used. For this purpose, a domain-specific behavioural analysis[2] is defined, and a case study [3] is specified and analysed. First, the generic analysis for checking whether the domain-specific behavioural property asserting that *when an event annotated with some tag is triggered, an event of some other tag must be triggerable in the future* is specified as a parameterised predicate operator. Second, a case study of the Automatic Teller Machine (ATM) (specified in section **??**) is modelled. The generic analysis is instantiated into a concrete analysis formalising the requirement **REQ4** (see Section 4.3), requiring that the *entered passcode must be followed by a confirmation or an abortion.* This section is organised as follows: the subsection 7.4.1 is dedicated to the definition of the domain-specific behavioural analysis including the definition of a domain ontology of events and the definition of a parameterised behavioural analysis. Subsection 7.4.2 is intended to specify the ATM case study and to check the behavioural analysis by applying a concrete version of the generic analysis predicate operator defined in subsection 7.4.1.

7.4.1 Defining a Domain-Specific Behavioural Analysis

Event Ontology Definition – (1) on Figure 7.2

The ontology modelling language (see section 5.2) is used to describe event tags which can be regarded as event classes. `input`, `confirmation`, and `finite` are important to the ATM case study since they will be involved in the formalisation of **REQ4**. The first two tags are used to denote respectively interaction events that provide user input information and formalise a user response. Finally, `finite` designates events that do not occur indefinitely.

```
CONTEXT  EventTagOntology
EXTENDS  ATMmEBModel
SETS  Tags , Ps
CONSTANTS
   eventOntology, tag, internal, preEmptive, nonPreEmptive, bounded, interaction, abortion,
   input, inputByVoice, inputByKeyboard, inputByScreen, confirmation,
   visualConfirmation, hapticConfirmation, auralConfirmation, textualConfirmation
AXIOMS
   Tags :
     partition(Tags, {tag}, {internal}, {preEmptive}, {nonPreEmptive}, {bounded}, {interaction},
     {input}, {inputByVoice}, {inputByKeyboard}, {inputByScreen}, {abortion}, {confirmation},
     {visualConfirmation}, {hapticConfirmation}, {auralConfirmation}, {textualConfirmation})
   eventOntology : eventOntology ∈ Ontology(Tags, Ps, Ev)
   classes : classes(eventOntology) = Tags
   properties : properties(eventOntology) = ∅
   classProperties : classProperties(eventOntology) = ∅
   instances : instances(eventOntology) = Ev
   classAssociations : classAssociations(eventOntology) = ∅
   instanceAssociations : instanceAssociations(eventOntology) = ∅
   EoIsWD : isWDOntology(eventOntology)
END
```

Listings 7.5: Context for event ontology instantiation

[2]The full Event-B theories are listed in appendix B.3
[3]The full Event-B models are listed in appendix C.3

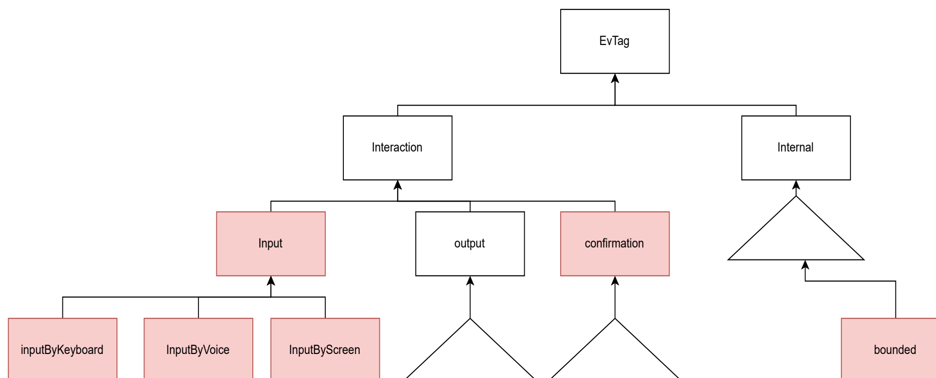


Figure 7.3: A tree representation of an ontology of events

Figure 7.3 depicts the graphical representation of the events classes and the inheritance relationships between them making up the ontology formally specified in Listing 7.5. This relation may be leveraged for the definition of analyses. Listing 7.5 (corresponding to `ontoContext` of Figure 7.2.(1)) contains the instantiation of the ontology modelling theory. It provides 3 type parameters: `Tags` for ontology classes, `Ps` for tag properties, and `Ev` for instances which are the model events (imported from `ATMmEBModel` context). The other ontology data type attributes are set to empty set here as they are not relevant to our development.

Behavioural Analysis Definition – (2) on Figure 7.2

The definition of the domain-specific behavioural analysis predicate is made of two abstraction layers. First, the predicate terms defining the analysis are specified and then these terms are used to build the predicate operator formalising the analysis. Higher-level predicate operators may be formed by the conjunction of these terms representing partial conditions of the analysis, and it shall be parameterised by the domain ontology. This subsection presents these two layers beginning with the higher-level which is used to achieve the actual analysis, and then detailed terms used for building the analysis are discussed.

The Presentation of The Higher-Level Analysis Predicate The domain-specific behavioural analysis defined for addressing requirement **REQ4** is formalised by the `isNecessarilyFollowedBy` predicate operator (see Listing 7.6).

The theory for expressing behavioural properties is presented in Listing 7.6. It corresponds to `predicate operator` in Figure 7.2.(2). The Event-B theory `Theo4Reachability` is where the lower-level terms composing the analysis are specified. The `EVENTT` is a special type parameter since it is used for instantiating the two theories; it plays both the role of events of the machine being analysed and the role of instances of ontology classes or tags. As previously

recommended for all operators, **REQ4** is formalised using two predicate operators: the well-definedness predicate operator `isNecessarilyFollowedByWD` and a predicate formalising the direct definition of the analysis. Indeed, the predicate `isNecessarilyFollowedBy` verifies whether *all* events annotated by tags in `srcTg` are **always** followed by some event annotated by tags in `trgTg` while passing through intermediate events exclusively annotated with `internalTg`. It relies on the predicate `Evt_Is_Always_Reachable_From_Definition` which formalises this reachability property on individual events.

```
THEORY BehaviouralPropertiesTheory
  IMPORT THEORY Theo4Reachability (importing EvtBTheo) ,  OntologiesTheory
  TYPE PARAMETERS STATEE, EVENTT, Tags, Ps
  OPERATORS
    isNecessarilyFollowedByWD < predicate > (m : Machine(STATEE, EVENTT) ,
        eo : Ontology(Tags, Ps, EVENTT) ,startTags : ℙ(Tags) ,transitTags : ℙ(Tags) ,
        endTags : ℙ(Tags) ,variants : ℙ(EVENTT × ℙ(STATEE × ℤ)))
      direct definition
        isWDOntology(eo)∧
        startTags ∪ transitTags ∪ endTags ⊆ getClasses(eo)∧
        startTags ∩ transitTags = ∅ ∧ endTags ∩ transitTags = ∅∧
        startTags ≠ ∅ ∧ endTags ≠ ∅ ∧ transitTags ≠ ∅∧
        (∀ti · ti ∈ startTags ∪ transitTags ∪ endTags
            ⇒ getInstancesOfClasses(eo, {ti}) ≠ ∅)∧
        variants ∈ getInstancesOfClasses(eo, startTags) → ℙ(STATEE × ℤ)∧
        (∀i · i ∈ getInstancesOfClasses(eo, startTags)
            ⇒ WD_reach(m, i,
                          getInstancesOfClasses(eo, endTags),
                          getInstancesOfClasses(eo, transitTags), variants(i)))
    isNecessarilyFollowedBy < predicate > (m : Machine(STATEE, EVENTT) ,
        eo : Ontology(Tags, Ps, EVENTT) ,startTags : ℙ(Tags) ,transitTags : ℙ(Tags) ,
        endTags : ℙ(Tags)   ,variants : ℙ(EVENTT × ℙ(STATEE × ℤ)))
      well−definedness
        isNecessarilyFollowedByWD(m, eo, startTags, transitTags, endTags, variants)
      direct definition
         ∀i · i ∈ getInstancesOfClasses(eo, startTags)
            ⇒ Evt_Is_Always_Reachable_From_Definition(
                    m, i,
                    getInstancesOfClasses(eo, endTags),
                    getInstancesOfClasses(eo, transitTags), variants(i))
END
```

Listings 7.6: `BehaviouralPropertiesTheory - isNecessarilyFollowedBy`

`isNecessarilyFollowedByWD` formalises the conditions for applying correctly the analysis. On the one hand, it checks that the ontology passed in as an argument is well-defined, that the classes of events make a partition, and that they are not empty. On the other hand, it checks that the instance of the Event-B meta-theory is well-built (see `WD_reach`), the target and intermediate events are progress events, and finally that the variant is well-defined for every source event. These conditions are important to ensure a meaningful interpretation of the analysis' outcomes. These two operators have six arguments:

- `m` a machine to be analysed,
- `eo` ontology to represent the domain concepts and constraints,
- `srcTg` a set of tags annotating the source events,
- `internalTg` a set of tags annotating the transit events,

- `trgTg` a set of tags annotating the target events, and
- `v` a list of variants associated with source events.

```
isPossiblyFollowedByWD < predicate > (m : Machine(STATEE, EVENTT) ,
      eo : Ontology(Tags, Ps, EVENTT) ,startTags : ℙ(Tags) ,transitTags : ℙ(Tags) ,
      endTags : ℙ(Tags) ,variants : ℙ(EVENTT × ℙ(STATEE × ℤ)))
   direct definition
      isWDOntology(eo)∧
      startTags ∪ transitTags ∪ endTags ⊆ getClasses(eo)∧
      startTags ∩ transitTags = ∅ ∧ endTags ∩ transitTags = ∅∧
      startTags ≠ ∅ ∧ endTags ≠ ∅ ∧ transitTags ≠ ∅∧
      (∀ti · ti ∈ startTags ∪ transitTags ∪ endTags
                 ⇒ getInstancesOfClasses(eo, {ti}) ≠ ∅)∧
      variants ∈ getInstancesOfClasses(eo, startTags) → ℙ(STATEE × ℤ)∧
      (∀i · i ∈ getInstancesOfClasses(eo, startTags)
                 ⇒ WD_reach(m,
                                   i,
                                   getInstancesOfClasses(eo, endTags),
                                   getInstancesOfClasses(eo, transitTags),
                                   variants(i)))
isPossiblyFollowedBy < predicate > (m : Machine(STATEE, EVENTT) ,
                                   eo : Ontology(Tags, Ps, EVENTT),
                                   startTags : ℙ(Tags), transitTags : ℙ(Tags), endTags : ℙ(Tags),
                                   variants : ℙ(EVENTT × ℙ(STATEE × ℤ)))
   well−definedness
         isPossiblyFollowedByWD(m, eo, startTags, transitTags, endTags, variants)
   direct definition
       ∀i · i ∈ getInstancesOfClasses(eo, startTags)
          ⇒ Evt_Is_Sometimes_Reachable_From_Definition(
                                        m,
                                        i,
                                        getInstancesOfClasses(eo, endTags),
                                        getInstancesOfClasses(eo, transitTags),
                                        variants(i))
```

Listings 7.7: `BehaviouralPropertiesTheory` - `isPossiblyFollowedBy`

It is noteworthy that `getInstancesOfClasses` returns a set of events annotated by a given set of tags (see Section 5.2, the Listing 5.2.3).

```
THEOREMS
WDNisWDP :
      ∀m, eo, startTags, transitTags, endTags, variants·
            (m ∈ Machine(STATEE, EVENTT)∧
            eo ∈ Ontology(Tags, Ps, EVENTT)∧
            startTags ∪ transitTags ∪ endTags ∈ ℙ(Tags)∧
            variants ∈ ℙ(EVENTT × ℙ(STATEE × ℤ)))
                  ⇒ (isNecessarilyFollowedByWD(
                     m, eo, startTags, transitTags, endTags, variants)
                        ⇒ isPossiblyFollowedByWD(
                           m, eo, startTags, transitTags, endTags, variants))
NisP :
      ∀m, eo, startTags, transitTags, endTags, variants·
            (m ∈ Machine(STATEE, EVENTT)∧
            eo ∈ Ontology(Tags, Ps, EVENTT)∧
            startTags ∪ transitTags ∪ endTags ∈ ℙ(Tags)∧
            variants ∈ ℙ(EVENTT × ℙ(STATEE × ℤ)))
                  ⇒ (isNecessarilyFollowedByWD(
                        m, eo, startTags, transitTags, endTags, variants)∧
                     isNecessarilyFollowedBy(
                        m, eo, startTags, transitTags, endTags, variants)
                           ⇒ isPossiblyFollowedBy(
                              m, eo, startTags, transitTags, endTags, variants))
```

Listings 7.8: `BehaviouralPropertiesTheory` - theorems

Listing 7.7 presents a weakened version of the analysis: given events possibly follow from other given events. This predicate relies on the operator imported from `Theo4Reachability`: `Evt_Is_Sometimes_Reachable_From_Definition`. The interest of this operator is to illustrate how different analyses may be organised through logical relationships. Indeed, as proved in Listing 7.8, if a model passes the necessary reachability analysis case then it will pass the possible reachability case. This may open the way to defining hierarchies of analyses.

The Presentation of The Detailed Analysis Terms The Event-B theory `Theo4Reachability` provides lower-level primitives for defining reachability analysis predicates in general, and they are particularly used for defining the domain-specific predicate operators previously discussed. It is noteworthy that this theory does not deal with domain-specific aspects; it is intended to define only temporal and reachability predicates.

```
THEORY Theo4Reachability
  TYPE PARAMETERS STATE, EVENT
  OPERATORS
    Evt__Is__Always__Reachable__From__Definition <predicate> (
        m : Machine(STATE, EVENT) , src : EVENT , trg : ℙ(EVENT) ,
        SubSetEvt : ℙ(EVENT) , variant : ℙ(STATE × ℤ))
      well−definedness WD_reach(m, src, trg, SubSetEvt, variant)
      direct definition
        NaturalVariant(m, variant, SubSetEvt)∧
        VariantDecrease(m, variant, SubSetEvt)∧
        Init_Local_Inv(m, src, Grd(m)[trg])∧
        Local_Inv_Preserved(m, src, SubSetEvt, Grd(m)[trg])∧
        Never_Exit_Transit_Zone(m, SubSetEvt,
                Progress(m) \ (SubSetEvt ∪ ({src} ∪ trg)), trg, variant)
    Evt__Is__Sometimes__Reachable__From__Definition <predicate> (
        m : Machine(STATE, EVENT) , src : EVENT , trg : ℙ(EVENT) ,
        SubSetEvt : ℙ(EVENT) , variant : ℙ(STATE × ℤ))
      well−definedness WD_reach(m, src, trg, SubSetEvt, variant)
      direct definition
        NaturalVariant(m, variant, SubSetEvt)∧
        VariantDecrease(m, variant, SubSetEvt)∧
        Init_Local_Inv(m, src, Grd(m)[trg])∧
        Local_Inv_Preserved(m, src, SubSetEvt, Grd(m)[trg])
END
```

Listings 7.9: `Theo4Reachability` - reachability predicates

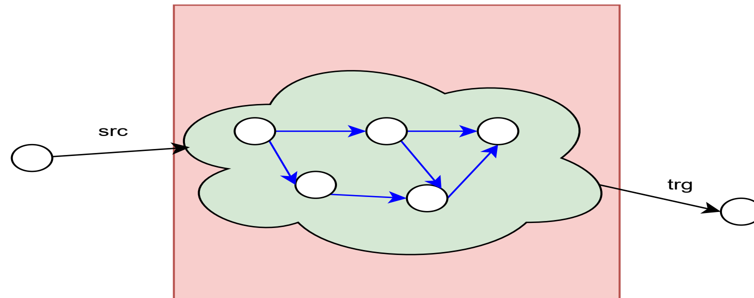


Figure 7.4: Simplified depiction of the necessary reachability analysis

```
Init_Local_Inv <predicate> (m : Machine(STATE, EVENT) ,src : EVENT ,
    lInv : ℙ(STATE))
  well−definedness Machine_WellCons(m) ,src ∈ Event(m)
  direct definition
    Get_next_act_state(m)(src) ⊆ lInv

Local_Inv_Preserved <predicate> (m : Machine(STATE, EVENT) ,
    InitEvent : EVENT ,SubSetEvt : ℙ(EVENT) ,LInv : ℙ(STATE))
well−definedness InitEvent ∈ Event(m) ∧ SubSetEvt ⊆ Progress(m)
direct definition
 ∀e · e ∈ SubSetEvt ⇒ BAP(m)[{e}][LInv ∩ Grd(m)[{e}]] ⊆ LInv

VariantDecrease <predicate> (m : Machine(STATE, EVENT) ,variant : ℙ(STATE × ℤ)
    ,SubSetEvt : ℙ(EVENT))
  well−definedness Inv(m) ◁ variant ∈ Inv(m) → ℤ ,Mch_INV(m) ,
      BAP_WellCons(m) ,SubSetEvt ⊆ Progress(m)
  direct definition
      ∀e, s · e ∈ Event(m) ∧ e ∈ SubSetEvt ∧ s ∈ State(m) ∧ s ∈ Inv(m)∧
         s ∈ Grd(m)[{e}] ⇒ (∀sp · sp ∈ BAP(m)[{e}][{s}]
             ⇒ (Inv(m) ◁ variant)(s) > (Inv(m) ◁ variant)(sp))

NaturalVariant <predicate> (m : Machine(STATE, EVENT) ,variant : ℙ(STATE × ℤ) ,
    SubSetEvt : ℙ(EVENT))
  well−definedness Inv(m) ◁ variant ∈ Inv(m) → ℤ ,BAP_WellCons(m) ,
      SubSetEvt ⊆ Progress(m)
  direct definition
    variant[Inv(m) ∩ Grd(m)[SubSetEvt]] ⊆ ℕ

Never_Exit_Transit_Zone <predicate> (m : Machine(STATE, EVENT) ,
    loopEvents : ℙ(EVENT) ,prohibtedEvents : ℙ(EVENT) ,trgEvent : ℙ(EVENT) ,
    variant : ℙ(STATE × ℤ))
  direct definition
   Get_next_states_of_evts(m)[loopEvents]∩
   Grd(m)[trgEvent] ∩ Grd(m)[prohibtedEvents]∩
   variant⁻¹[ℕ] = ∅
```

Listings 7.10: `Theo4Reachability` - low-level terms

In particular, the `Evt_Is_Always_Reachable_From_Definition` operator is specified as a conjunction of primitives predicates covering three important conditions. Figure 7.4 depicts the three conditions (1) the green region indicates that a given invariant is preserved denoting states attainable by the allowed intermediate events, (2) the red region depicts the states which are not reachable by the allowed intermediate events, and finally (3) the blue arrows in the green region indicate that the allowed events need to decrease a given variant. Indeed, as shown in Listing 7.10 (`Init_Local_Inv`) and `Local_Inv_Preserved` predicates formalise the condition (1) where the local invariant is taken for simplicity as the state set represented by the guard of the target events `Grd(m)[{trg}]`. This ensures that the target events are always triggerable. The `VariantNatural` and `VariantDecrease` predicates ensure that the system will not be caught in a loop of the allowed intermediate events; this amounts to condition (3).

The condition (2) is formalised by the predicate `Never_Exit_Transit_Zone` which checks that no transition exists to the states not contained in the given invariant. These three conditions completely formalise the requirement that a specified target event is *always* reachable from a given source event by triggering only a given set of events.

The analysis is strict in the sense that it requires that the events imply

the guards of the target events; this means that all attainable states of the intermediate events shall imply the guard of the target events.

Without a loss of generality, we can apply this analysis to systems not satisfying this condition with the help of refinement. Indeed by refinement, we can address systems that reach the wanted states after many steps; the refinement of the intermediary events does not break the analysis.

```
One__Next__Evt__Is__Triggerable <predicate> (m : Machine(STATE, EVENT),
    variant : ℙ(STATE × ℤ), SubSetEvt : ℙ(EVENT))
  well−definedness Inv(m) ◁ variant ∈ Inv(m) → ℤ  , BAP_WellCons(m),
      SubSetEvt ⊆ Progress(m) ∧ Mch_INV(m)
  direct definition
     ∀e, s · e ∈ SubSetEvt ∧ s ∈ BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]]∧
     (Inv(m) ◁ variant)(s) ∈ ℕ ⇒ s ∈ Grd(m)[SubSetEvt]

Is__Src__Next__Trg <predicate> (m : Machine(STATE, EVENT), src : EVENT,
    trg : EVENT)
  well−definedness BAP_WellCons(m), {src, trg} ⊆ Progress(m)
  direct definition
    BAP(m)[{src}][Inv(m) ∩ Grd(m)[{src}]] ⊆ Grd(m)[{trg}]

WD__reach <predicate> (m : Machine(STATE, EVENT), src : EVENT,
    trg : ℙ(EVENT), SubSetEvt : ℙ(EVENT), variant : ℙ(STATE × ℤ))
   direct definition
    Machine_WellCons(m) ∧ trg ⊆ Progress(m) ∧ src ∈ Event(m)∧
    Inv(m) ◁ variant ∈ Inv(m) → ℤ∧
    Mch_INV(m) ∧ SubSetEvt ⊆ Progress(m)
```

Listings 7.11: `Theo4Reachability` - auxiliary predicates

`Evt_Is_Sometimes_Reachable_From_Definition` is quite similar to the other operator except that it does not require the exit condition (`Never_Exit_Transit_Zone`). Several auxiliary operators are defined to support the definition of reachability predicate analysis. Listing 7.11 presents an extract of such a collection of utility operators that are used in the definition of primitive condition operators. In particular, `WD_reach` groups different important conditions of well-definedness about machines and variants. Two other operators `Is_Src_Next_Trg` and `One_Next_Evt_Is_Triggerable` are defined where the first operator states that a `trg` event is always triggerable when `src` is activated. The second predicate operator ensures that when the variant is still positive and non-zero there is always at least one allowed intermediate event that is activable.

7.4.2 Applying a Domain-Specific Behavioural Analysis

This subsection illustrates how previously formalised analysis may be used to investigate a formal model of the ATM's user interface, whose informal description is presented in section 4.3.

Exporting Models to Event-B meta-theory – (3) on Figure 7.2

The Event-B model of the user interface is first presented. Next, the translation of this model to obtain an instance of the Event-B meta-theory is given. The Event-B modelling of the ATM case study is composed of a context and a

machine. The context `ATMEnvironment` introduces carrier sets, constants and states basic facts as axioms for modelling static parts of the system.

Specifically, Listing 7.12 presents numerous enumerations and axioms. Indeed, `INSERTION_STATUS` represents the state of the card: whether it is inserted or not. `INPUT_MODE` contains two constants for indicating which input device is used: keyboard or screen. It allows addressing **REQ1**. Several constants are introduced to address the brightness feature of the system, in particular, `BRIGHTNESS_MIN` and `BRIGHTNESS_MAX` specify the minimum and maximum levels of brightness allowed. Axiom `axm1` allows to address **REQ6** by introducing a maximum number of attempts `MAX_ATTEMPTS`. Also, constants are defined for handling the strings of entered passcodes. All possible passcodes are formalised by the carrier set `STRINGS`. In addition, `CORRECT_PASSCODE` and `EMPTY_STRING` represent specifically the correct passcode and a default value for initialising the buffers containing the strings. Finally, `AMOUNTS` represents a set of all possible deliverable cash by the ATM, and `NO_MONEY` defines a default value meaning that no cash will be delivered.

CONTEXT
ATMEnvironment
SETS *INPUT_MODE*, *INSERTION_STATUS*, *STRINGS*, *AMOUNTS*
CONSTANTS
MAX_ATTEMPTS,
CORRECT_PASSCODE,
KEYBOARD,
SCREEN,
IN,
OUT
BRIGHTNESS_LEVELS,
BRIGHTNESS_MIN,
BRIGHTNESS_MAX,
EMPTY_STRING,
MAX_BRIGHTNESS_UPDATE,
NO_MONEY
AXIOMS
$axm1 : MAX_ATTEMPTS \in \mathbb{N}1$
$axm2 : CORRECT_PASSCODE \in STRINGS \land EMPTY_STRING \in STRINGS$
$axm3 : CORRECT_PASSCODE \neq EMPTY_STRING$
$axm4 : partition(INPUT_MODE, \{KEYBOARD\}, \{SCREEN\})$
$axm5 : partition(INSERTION_STATUS, \{IN\}, \{OUT\})$
$axm6 : BRIGHTNESS_MIN \in \mathbb{N}$
$axm7 : BRIGHTNESS_MAX \in \mathbb{N}$
$axm8 : BRIGHTNESS_MAX > BRIGHTNESS_MIN$
$axm9 : BRIGHTNESS_LEVELS = BRIGHTNESS_MIN..BRIGHTNESS_MAX$
$axm10 : MAX_BRIGHTNESS_UPDATE \in \mathbb{N}$
$axm11 : NO_MONEY \in AMOUNTS$
$axm12 : AMOUNTS \setminus \{NO_MONEY\} \neq \varnothing$
END

Listings 7.12: ATM case study - contextual information

The second part of the model is the Event-B machine presented in Listing 7.13. Invariant `inv11` formalising **REQ5** (see section 4.3) specifying that the entered passcode is never displayed. The events styled in italic font are highlighted since they are involved in the verification of the requirement **REQ4**. The goal of the analysis is to prove in particular that `KBDString` will be necessar-

ily followed by `ConfirmKBDString` while allowing a fixed number of activation of event `changeBrithness`. The property shall be verified for `SCRString` and `confirmSRC`. The analysis will be carried out on an exported version of this model after annotation.

```
MACHINE ATMUserInterface
SEES ATMEnvironment
VARIABLES  string , virtualNumpadRegister , keyboardRegister , attempts ,
           confirmationStatus , validationStatus , deliveryStatus , isStringVisible ,
           inputMode , cardStatus , brightness , brightnessUpdates , newString , sum
INVARIANTS
   ...
   inv11 : string = virtualNumpadRegister ∨ string = keyboardRegister
   inv12 : isStringVisible = FALSE
   inv13 : brightness ∈ BRIGHTNESS_LEVELS
   inv14 : brightnessUpdates ∈ ℤ
   ...
EVENTS
   INITIALISATION...

   insertCard...

   KBDString
   WHERE
      grd1 : 0 ≤ attempts ∧ attempts < MAX_ATTEMPTS
      grd2 : inputMode = KEYBOARD
      ...
   THEN
   act1 : string, keyboardRegister :| keyboardRegister′ ∈ STRINGS ∧ string′ = keyboardRegister′
   act2 : brightnessUpdates := 0
      ...
   END

   SCRString...

   changeBrightness...
   WHERE
      grd1 : brightnessUpdates < MAX_BRIGHTNESS_UPDATE
      ...
   THEN
      act1 : brightness :∈ BRIGHTNESS_LEVELS
      act2 : brightnessUpdates := brightnessUpdates + 1
   END

   confirmKBDString
   WHERE
      grd1 : 0 ≤ attempts ∧ attempts < MAX_ATTEMPTS
      grd2 : inputMode = KEYBOARD
      ...
   THEN
      act1 : attempts := attempts + 1
      act2 : confirmationStatus := TRUE
   END
   confirmSRCtring
   ...
END
```

Listings 7.13: ATM case study - Event-B model

The export operation is straightforward ; it consists in instantiating the data type `Machine(St, Ev)`: Event-B machine can be formalised as an instance of the Event-B meta-theory (Machine $\mathtt{M}_i$ on Figure 7.2.(A)). To define an Event-B machine as an instance, it is enough to instantiate (give values) to the `Machine(St, Ev)` attributes at instantiation (see Listing 7.1). The

`St` type parameter is substituted by a Cartesian product of the set types of `ATMUserInterface` machine state variables (14 in total) and `Ev` by the set of the events of this machine.

```
CONTEXT
   ATMmEBModel
EXTENDS
   ATMEnvironment
SETS
   Ev
CONSTANTS
   ATM
   init, insertCard, KBDString, SCRString, changeBrightness, confirmKBDString
   confirmSCRString, checkStringCorrect, checkStringWrong, deliverBankNotes
AXIOMS
   Ev : partition(Ev, {init}, {KBDString}, {changeBrightness}, {confirmKBDString}, ...,
          {checkStringWrong}, {deliverBankNotes})
   ATM : ATM ∈ Machine(STRINGS × STRINGS × STRINGS × ℤ × BOOL × BOOL×
                        BOOL × BOOL × INPUT_MODE × INSERTION_STATUS×
                        ℤ × ℤ × BOOL × AMOUNTS, Ev)
   Event : Event(ATM) = Ev
   Thm : Thm(ATM) = State(ATM)
   Grd : Grd(ATM) = {e ↦ (string ↦ virtualNumpadRegister ↦ keyboardRegister ↦
            attempts ↦ confirmationStatus ↦ validationStatus ↦ deliveryStatus ↦
            isStringVisible ↦ inputMode ↦ cardStatus ↦ brightness ↦
            brightnessUpdates ↦ newString ↦ sum) |
         (e = KBDString ∧ 0 ≤ attempts ∧ attempts < MAX_ATTEMPTS∧
         inputMode = KEYBOARD ∧ cardStatus = IN ∧ newString = FALSE)∨
         (e = changeBrightness ∧ brightnessUpdates ≤ MAX_BRIGHTNESS_UPDATE∧
         cardStatus = IN)∨
         (e = confirmKBDString ∧ 0 ≤ attempts∧
            attempts < MAX_ATTEMPTS ∧ inputMode = KEYBOARD ∧ cardStatus = IN∧
            confirmationStatus = FALSE ∧ validationStatus = FALSE ∧ newString = TRUE)∨
            ...
   BAP : BAP(ATM) = {e ↦ ((string ↦ ... ↦ sum) ↦ (stringp ↦ ... ↦ sump)) |
              (e = KBDString ∧ keyboardRegisterp ∈ STRINGS∧
                 stringp = keyboardRegisterp ∧ brightnessUpdatesp = 0 ∧ newStringp = TRUE∧
                 confirmationStatusp = FALSE ∧ validationStatusp = FALSE ∧ ...)∨
              (e = confirmKBDString ∧ attemptsp = attempts + 1∧
              confirmationStatusp = TRUE ∧ ...∨
              (e = changeBrightness ∧ brightnessp ∈ BRIGHTNESS_LEVELS∧
                 brightnessUpdatesp = brightnessUpdates + 1 ∧ ...)∨
                 ...
   check_Machine_Consistency :
      check_Machine_Consistency(ATM)
END
```

Listings 7.14: ATM case study - Event-B meta-theory instance

Listing 7.14, corresponding to $\mathtt{M}_i$`Context` in Figure 7.2.(3), shows an extract of the `ATMUserInterface` machine exported as an instance of the Event-B meta-theory. Guards and actions of the events are formalised, as instances, in the `Grd(ATM)` and `BAP(ATM)` sets (axioms `axm4` and `axm5`).

Annotation & analysis – (4) on Figure 7.2

The final step before checking the behavioural property of necessary precedence is *annotation*. The annotation allows linking the domain knowledge concepts and constraints to the design model. This is achieved by assigning events to corresponding tags. For example, the `KBDString` is assigned to `textualConfirmation`, `confirmation` and `Tag` – see `annotateDef` in List-

ing 7.15. The assignment shall respect the subclass relationship between tags. Therefore an event shall be assigned to all super-classes of its real class.

```
CONTEXT
    AnnotatedModel
EXTENDS
    EventTagOntology
CONSTANTS
    annotate
    variants
AXIOMS
  annotateDef : annotate =
      ({bounded} × {changeBrightness})∪
      ({inputByKeyboard} × {KBDString})∪
      ({inputByScreen} × {SCRString})∪
      ({inputByVoice} × ∅)∪
      ({input} × {KBDString, SCRString})∪
      ({textualConfirmation} × {confirmKBDString}) ∪ ...
  classInstances : classInstances(eventOntology) = annotate
  isWDOntologyThm : isWDOntology(eventOntology)
  variantDef : variants =
        {KBDString ↦
            {p ↦ bright ↦ ck ↦ cs ↦ v |
              p ∈ STRINGS × STRINGS × STRINGS × ℤ×
                BOOL × BOOL × BOOL × BOOL×
                INPUT_MODE × INSERTION_STATUS × ℤ ∧ bright ∈ ℤ∧
              ck ↦ cs ∈ BOOL × AMOUNTS∧
              v = MAX_BRIGHTNESS_UPDATE − bright}}∪
        {SCRString ↦ ...}
  vThm : variants ∈ annotate[{input}] → ℙ(State(ATM) × ℤ)
  isNecessarilyFollowedByThm :
   isNecessarilyFollowedBy(ATM, eventOntology, {input}, {bounded}, {confirmation}, variants)
END
```

Listings 7.15: ATM case study - annotation and analysis context

The correct verification of the analysis `isNecessarilyFollowedBy` requires theses following conditions as illustrated in Listing 7.15:

1. The model events are annotated using the `annotate` relation as stated by the axiom `annotateDef`,
2. They are related to the `eventOntology` via `classInstances`,
3. The ontology shall be verified by proving `isWDOntologyThm`,
4. Variants shall be associated with every source event: `KBDString` and `SRCString` as asserted by `variantDef`, furthermore `vThm` ensures that it is well-defined.

The last theorem `isNecessarilyFollowedByThm` represents the analysis verification on the `ATMUserInterface` model, it states that the requirement **REQ4** is satisfied. It is noteworthy that the preparatory step is guided by the well-definedness condition of the analysis operator. Indeed, the WD is structured so that this individual conditions may be identified. For example, theorems `isWDOntologyThm` stating that the ontology is well-defined, and `vThm` stating that the variant is well-defined are identified based on the well-definedness conditions of `isNecessarilyFollowedBy` predicate operator.

7.5 Advantages of the Framework

This section discusses the advantages of the framework described in Section 7.3 corroborated by the observations of Section 7.4. Numerous benefits may be stated for the framework, as presented below.

7.5.1 Principled Approach and Reusability

The primary objective of the framework was to define an approach that can be used for describing and applying *domain-specific* behavioural analyses. This goal is successfully achieved, and it was illustrated by a critical case study. The approach relies on two theories presented respectively in [116] and [143]: (1) the ontology modelling language which allows defining domain knowledge as an ontology, and (2) the Event-B meta-theory modelling language which provides a way to reason on Event-B concepts. This approach overcomes the drawbacks of the shallow approaches consisting of incorporating the analysis requirements into the model at the same level as the system requirements. Furthermore, the framework allows the reusability of analyses and facilitates their sharing. There are two levels of reusability: (1) the ability to apply the analysis on several system models, and (2) the ability to reuse the same analysis parameterised with different domain knowledge models. The two kinds of reusability are supported thanks to the fact that the analysis is formalised as a generic predicate operator parameterised with (1) the system model as an instance of Event-B meta-theory, and (2) the domain model as an instance of the Ontology Modelling Language Event-B Theory. The framework was successfully used to describe a particular domain-specific behavioural analysis (verifying that *input* events are necessarily followed by *confirmation* events). Moreover, this particular analysis has been used to verify that a specified requirement (**REQ4**) is fulfilled.

7.5.2 Non-intrusiveness

The application of the analyses defined using the framework is not intrusive since they rely on the annotation of the model (exported as an instance of Event-B) as depicted by step (3) on Figure 7.2. This feature of the framework has been demonstrated in the case study where the original Event-B model shown in Listing 7.13 has been translated to an instance of the Event-B meta-theory (the result is illustrated by Listing 7.14).

7.5.3 Verification Based on Theorem Proving

The approach uses proof-based verification to check that a domain-specific behavioural analysis applies successfully to models. Indeed, the analysis is established by a theorem proving where the predicate operator formalising the analysis is discharged. The proof paradigm has a number of advantages, in particular, it does not suffer from the state exposition problem inherent to model-checking techniques. This has been illustrated in Listing 7.15 where the

theorem `isNecessarilyFollowedByThm` is discharged to establish the requirement **REQ4** for the ATM model.

In contrast with this approach, consider another approach consisting of boiling down the domain knowledge behaviour properties into temporal properties and then using model checking to verify the resulting set of temporal properties. The model is passed as an argument to the analysis procedure alongside the domain knowledge model. This approach meets quickly its limitation when the systems are large and complex due to the classical problem of state explosion. Furthermore, the manual translation of domain knowledge to temporal formulas is tedious.

7.5.4 Proof and Modelling Effort Reduction

The proposed approach reduces proof and modelling effort. Indeed, using the framework allows for a significant reduction in proving effort in the long run. This is true because redundant proofs such as the proof of the well-definedness of an analysis are done once and for all. Moreover, all the proofs related to generic components are discharged only once. Specifically, theorems of `OntologiesTheory`, `EvtBTheo` and `BehaviouralPropertiesTheory` do not require to be reproved for every analysis application since they are proved at the definition time. Additionally, the architecture of the framework (based on generic Event-B theories) permits the definition of a collection of lemmas useful for establishing the analysis of the models. Similarly, the generic components of the framework are modelled once and for all. This is the case for `OntologiesTheory` and `EBTheory`, but also for the theory formalising the analysis `BehaviouralPropertiesTheory`; it corresponds to part **(B) Theories** on Figure 7.2. Only the components (Event-B contexts) in the part **(C) Instance** on Figure 7.2 need to be modelled and proved.

Event-B Models and Theories	Number of proof obligations
OntologiesTheories	21
EvtBTheory + EvtBManip	10
Theo4Reachability	16
BehaviouralPropertiesTheory	7
ATMEnvironment	0
EventTagOntology	1
ATMUserInterface	16
ATMmEBModel	7
AnnotatedModel	4

Table 7.1: Proof statistics of behavioural analysis and ATM case study

Table 7.1 shows a summary of the proof obligations generated and discharged for different theories and models, which are developed for our approach and the illustrative case study. Proof obligations of theories created at definition time are proved only once. This is the case for `OntologiesTheories`,

`EvtBTheory`, `EvtBManip` and `Theo4Reachability`: they were imported and reused from existing projects, so re-proving was not necessary. However, the theory `BehaviouralPropertie sTheory` providing the domain-specific behavioural analysis predicates required to prove several well-definedness conditions related to the two analyses: the *necessary* case predicate `isNecessarilyFollowedBy` and the *possible* case predicate `isPossiblyFollowedBy`. The `EventTagOntology` context requires to prove at least one theorem ensuring that the created ontology is well-defined. Finally, all the proof effort concentrates on the modelling of the system corresponding to `ATMUserInterface` machine and the application of the analysis corresponding to `ATMmEBModel` Event-B meta-theory instance (context) and `AnnotatedModel` context. Indeed, the most difficult proof relies in verifying the theorem of the analysis. Interestingly, the proof of the analysis may reveal errors in the model or less probably in the analysis itself. Therefore, proving the analysis predicate operator may help correct errors in the model.

7.5.5 Generalisation

The approach has been successfully applied to systems belonging to the domain of interactive critical systems where an ontology for annotating interaction events is defined. Indeed, this ontology has been taken into account in the definition of behavioural analysis. It is noteworthy that this approach is general in the sense that it may be used to analyse different systems such as railway, medical or other systems provided that the underlying domain is formalised as an ontology.

7.6 Conclusion

This chapter addressed the issue of analysing domain-specific behavioural properties of systems. An integrated framework centred around the Event-B method was proposed for investigating non-intrusively behavioural properties mined from domain knowledge. It was based on the ontology modelling language introduced in section 5.2, and the Event-B meta-theory presented in section 7.2. The proposed approach has 4 steps (see section 7.3):

1. formalising domain knowledge as an ontology,
2. defining domain-specific behavioural analyses,
3. exporting the model to the reflexive Event-B framework EB4EB to manipulate the Event-B concepts such as events and their components,
4. annotating and analysing Event-B models through theorem proving.

The approach is illustrated, in section 7.4, through a concrete analysis checking that *a given class of events are necessary followed by another class of events meanwhile only third class of events are transitioned*. This analysis was used to check a behavioural requirement (**REQ4**) of the ATM case study (see section 4.3 for more details on the ATM case study).

Chapter 8

Formal Conformance Checking

This Chapter contains:

8.1 Introduction 116
8.2 Our approach 116
8.2.1 A Standard Formal Specification —(2) on Figure 8.1 . . . 117
8.2.2 Standard Theory Instantiation —(3) on Figure 8.1 118
8.2.3 Model Annotation —(4) on Figure 8.1 119
8.3 Formalisation of ARINC 661 Standard 119
8.3.1 `ARINC661Theory` - Concepts Declaration 120
8.3.2 `ARINC661Theory` - Operators Declaration 121
8.3.3 `ARINC661Theory` - Primitives Definitions 122
8.3.4 `ARINC661Theory` - Theorems 124
8.4 Weather Radar Application Case Study 124
8.4.1 `WXRTheory` - Instances Declaration 125
8.4.2 `WXRTheory` - Instances Definition 125
8.4.3 `WXRTheory` - Operators Declaration and Definition 126
8.4.4 `WXRTheory` - Theorems 127
8.4.5 Annotated Model of WXR —(4) on Figure 8.1 128
8.5 Advantages of The Framework 130
8.5.1 Achieving Standard Conformance Formally 130
8.5.2 Qualitatively Enhanced System Models 131
8.5.3 Reduction of Modelling and Proving Effort 131
8.5.4 Enabling Evolution of Standard 132
8.6 Conclusion 132

Verifying the standard conformance of a system design is a necessary task in the system engineering life cycle, in particular when the said system is deemed critical. Standard compliance verification means ensuring that a system or a model of a system faithfully meets the requirements of a standard, in particular

domain and certification standards. It aims to improve the robustness and trustworthiness of the system model.

In this chapter, an approach for achieving formal conformance to standards is presented. The conformance-checking process allows the establishing of safety properties required by the specification document of certification standard. The framework presented in chapter 6 is considered the bedrock upon which the definition of the conformance-checking approach relies. It is qualified as *conformance-by-construction*, since the system model uses the primitives of the formalised standard, and it inherits the properties required by the standard. The fine-grained details are discussed in Section 8.1. This framework facilitates the formalisation of standard concepts and rules as an ontology, as well as the formalisation of an engineering domain. Conformance checking is accomplished by annotating the system model with typing conditions. An industrial case study is proposed, borrowed from the aircraft cockpit engineering domain to demonstrate the feasibility and strengths of the approach. First, the ARINC 661 certification standard is used; it is formalised as an Event-B theory. Secondly, the weather radar system (WXR) user interface is used as a case study; it was checked against the requirements specified by the ARINC 661 certification standard. Sections 8.3 and 8.4 present the formal modelling in Event-B of the ARINC 661 standard and the WXR application respectively.

8.1 Introduction

Checking the conformance of system design models and/or implementation to a standard is often achieved by informal or semi-formal processes like argument-based reports produced through model reviews, testing and simulation, experimentation, and so on [1]. Conventionally, these qualification methods have proven to be valuable for system engineering in areas like transportation systems, medical devices, power plants, etc. Yet, formal checking of conformance, as advocated by the DO178-C, allows for higher trustworthiness and reliability. Among them, extensive case coverage and availability of automatic verification capabilities such as theorem proving and model-checking. In this chapter, the properties specified by standards known to be safety properties, are targeted.

8.2 Our approach

The framework for achieving formal conformance of models to standards is depicted in Figure 8.1. It is composed of three phases: *Conceptualisation*, *Instantiation*, and *Annotation*. These phases are different, and they are not repeated for each development. In particular, the conceptualisation is done only once for every standard. For modelling a system, the phases are carried out in the following order:

- **Conceptualisation**: a standard specification is formalised such that the concepts and constraints interpreted from the standard specification doc-

ument are expressed as theories **(2)** in terms of data types and operators. It is noteworthy that using the plain Event-B theories would result in heterogeneous formalisation, the ontology modelling language is used as modelling language —`OntologiesTheory` is represented by **(1)** on the Figure 8.1, and it is discussed in section 5.2.

- **Instantiation**: the requirements of the system being designed are handled by instantiating the theories developed for formalising the standard **(3)**,
- **Annotation**: system model is annotated through typing **(4)** and associated operator are exclusively used. This annotation allows transferring the constraints and rules, expressed as theorems, to the design model establishing standard conformance.

Note that `OntologiesTheory` **(1)** is defined once and for all. The theory formalising the standard concepts, rules and properties **(2)** are formalised in stable theories which may evolve as the standard gets updated in new releases. This evolution of standards may or may not affect the validity of the theorems of the theory —if the updated theory still entails the theorems then the models depending on this theory remain valid from the verification point of view. In Figure 8.1, `Instantiates` and `Imports` links correspond to Event-B built-in constructs (generic type parameters instantiation is automatically achieved by type synthesis), and `Annotation` is implemented by typing model concepts with theories data types using the `Sees` Event-B construct.

A key requirement to set up our approach is the exclusive use of data types and operators provided by the Event-B theory formalising the standard specification. This condition is necessary to ensure that theorems entailed by operators are transferred and therefore provable in the Event-B model. A comprehensive study of these techniques is presented in [13].

In the following subsections, the emphasis is put on formal specification of standards and the formal verification of standard conformance, which correspond to phases **(2)**, **(3)** and **(4)** on Figure 8.1. These phases are described below.

8.2.1 A Standard Formal Specification —(2) on Figure 8.1

The first phase of the approach consists in specifying formally the standard using the ontology modelling language `OntologiesTheory` (see section 5.2). Type parameters `C`, `P` and `I` are instantiated with the standard's objects and object's attributes respectively. Furthermore, the standard's rules and constraints are formalised as a set of axioms and/or embedded in the definition of the operators.

Conformance criteria and the properties of the standard that are required for the objects are formalised and proved as a set of theorems. Indeed, the operators must entail these required properties. One important benefit is that these proofs are carried out once and for all, and may then be transferred to models constructed using the standard data types and operators.

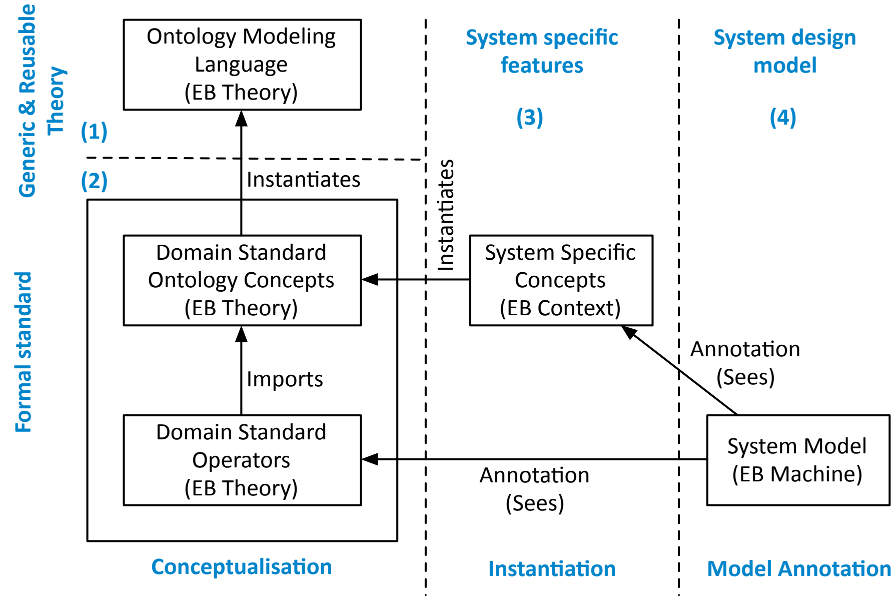


Figure 8.1: The framework for standard conformance-by-construction

8.2.2 Standard Theory Instantiation —(3) on Figure 8.1

At this phase, the elements specific to the system being modelled are defined; the Event-B theories formalising the standards are used.

The classes are associated with their instances and the associations between instances are specified taking into account the well-definedness conditions required by ontology instantiation, i.e. `isWDgetInstancePropertyValues`. Three attributes of the ontology are completed by theory instantiation: `instances`, `classInstances` and `instancePropertyValues`.

The definitions of these components are *system-dependent* and represent the elements of the system as instances of the standard classes and class associations. The component of the ontology depending on the instances type parameter is defined at this point.

The `CheckOfSubsetOntologyInstances` operator is used to ensure that system-specific concepts comply with defined standard ontology. The validity of the instantiation is checked using the predicate operator `isWDOntology` which verifies whether the instantiation of the three remaining attributes of the ontology keeps the ontology definition correct. In particular the predicate `isWDInstanceAssociations`, which is a clause of the `isWDOntology` predicate, must be verified to ensure that the set of triples (instance, properties and value) do comply with the classes schema specified in `isWDClassAssociations`. The role of these operators is explained in more detail in Section 5.2.

8.2.3 Model Annotation —(4) on Figure 8.1

After the formal specification and definition of standards, the next phase is model annotation. It consists of typing model variables with instance-related ontology components, generally `instancePropertyValues`, to comply with data types provided by the formalised standard.

A key methodological requirement in the description of the formal model is to use operators provided by the theory formalising the standard exclusively. It is therefore valid to state that the theorem of the theory formalising the standard is also the theorems of the models, i.e. they are invariants of the model.

In Event-B, this means that the formalised standard requirements and safety properties are expressed as theorems and the proof obligations are discharged deductively using the theory theorems and the working hypothesis of exclusively using the theory operators. Indeed, this assertion necessitates that the model transitions are realised, *exclusively*, with the operators provided by the theory describing the domain standard.

Furthermore, operators' application produces well-definedness proof obligations requiring to prove the correctness of the application, such properties are discharged, by ensuring that the guards are strong enough to imply the well-definedness of the operators or their combination. In Section 8.4, a model of the user interface of a critical system is used to illustrate how the property transfer is achieved.

8.3 Formalisation of ARINC 661 Standard

ARINC 661 [18] is the Cockpit Display System (CDS) standard for communication protocols between interface objects and aircraft systems. It has been used for the development of interactive applications in, for instance, the Airbus A380 and the Boeing B787 [1].

In ARINC 661 specification standard, an interactive application is called a User Application (UA), that receives input from the CDS and triggers actions in aircraft systems. Such inputs are produced by the flying crew manipulating specific input devices such as a KCCU (Keyboard Cursor Control Unit). UAs also receive information flow from aircraft systems that are presented to the flying crew using interactive objects whose behaviour and parameters are described in the standard. The current version of the standard (called supplement 7 for part 1) describes, in about 800 pages, a set of definitions and requirements for the CDS and its graphical objects (called widgets).

A key task in addressing formal conformance is to formally express the base standard which is the reference in the conformance-checking process. Hereafter, phase (2) of the approach presented in Section 8.2 is showcased through the formalisation of a part of ARINC 661. `ARINC661Theory`[2] is an Event-B theory that embodies the formal definition of a part of the ARINC 661 stan-

[1] https://www.presagis.com/en/product/arinc-661/
[2] The full listing of `ARINC661Theory` is in appendix B.2

ARINC 661 element	Reference (page)	Event-B formal element
Label	3.3.20 (p114)	Label
RadioBox	3.3.34 (p184)	RadioBox
CheckButton	3.3.5 (p80)	CheckButton
SELECTED, UNSELECTED	3.3.5-1 (p81)	SELECTED, UNSELECTED
CheckButtonState	3.3.5-1 (p81)	hasCheckButtonState
LabelString	3.3.5-1(p81)	hasLabelStringForCheckButton
Textual paragraph	3.3.34 (p185)	isWDRadioBox
...	...	...

Table 8.1: Mapping between ARINC 661 concepts and Event-B formalisation

dard as defined in the specification document (see [18] for a complete description). `ARINC661Theory` is expressed using the ontology modelling languages `OntologiesTheory`. The definition of the ARINC 661 Event-B theory is composed of four main parts: the concept declaration part, the operator declaration, the axiomatisation part which gives the definitions of the concepts and the operators, and finally, the part enumerating the theorems formalising the required properties the standard is intended to enforce.

8.3.1 `ARINC661Theory` - Concepts Declaration

An in-depth analysis of the ARINC 661 standard specification document [18], especially the part describing the library of widgets allows us to identify the hierarchical organisation of the widget. They are straightforwardly amenable to ontological structure formalised using `OntologiesTheory`. The widget may be seen as a concept with a collection of attributes. The concepts may also share a set of associations.

First, the ARINC 661 widgets are directly formalised as classes of the ontology and their attributes are formalised as properties of the classes. For example, Label and check button states are respectively formalised as `Label` class and `hasCheckButtonState` property. Second, the rules and constraints are expressed as axioms, for example, the `isWDRadioBox` is a predicate operator axiomatising the constraint that the button children of a radio box may be selected exclusively. Table 8.1 shows identified correspondences between ARINC 661 concepts and their formal counterparts with an exact location of the description in the standard specification document.

The formalisation is guided by the structure of the ARINC 661 widget library and is expressed in the ontology modelling language formalised in Event-B theory `OntologiesTheories`. `C`, `P` and `I` of `OntologiesTheories` are instantiated by three abstract types: `ARINC661Classes`, `ARINC661Properties` and `ARINC661Instances`. `ARINC661Classes` contains the classes formalising the widget types specified by ARINC 661 standard like labels, check buttons and radio boxes, etc.

The Listing 8.1 enumerates the constants (constant theory operators) playing the role of classes, properties and instances of the ontology.

- `Label`, `RadioBox` and `CheckButton` are classes of the ontology.

- `ARINC661_BOOL` is a class that represents the boolean type and has only two instances `A611_TRUE` and `A661_FALSE`.
- `hasCheckButtonState` is an ontology property that represents the attribute of the ARINC 661 widget type check button.

The attributes of the ARINC 661 widget types are defined as ontology properties and are associated with their respective widgets through the ontology component `classProperties`.

```
THEORY ARINC661Theory
  IMPORT THEORIES OntologiesTheory
  AXIOMATIC DEFINITIONS  ARINC661Axiomatisation :
    TYPES ARINC661Classes, ARINC66Properties, ARINC661Instances
    OPERATORS
      ARINC661_BOOL < expression > () : ARINC661Classes
      ARINC661_BOOL_EXTENDED < expression > () : ARINC661Classes
      A661_TRUE < expression > () : ARINC661Instances
      A661_FALSE < expression > () : ARINC661Instances
      RadioBox < expression > () : ARINC661Classes
      CheckButton < expression > () : ARINC661Classes
      hasChildrenForRadioBox < expression > () : ARINC66Properties
      hasCheckButtonState < expression > () : ARINC66Properties
      SELECTED < expression > () : ARINC661Instances
      UNSELECTED < expression > () : ARINC661Instances
      ...
```

Listings 8.1: `ARINC661Theory` - constants

8.3.2 `ARINC661Theory` - Operators Declaration

Furthermore, predicate operators are defined for formalising the constraints and rules of the ARINC 661 standard. The Listing 8.2 shows only the declarations of such operators. For example, `isWDRadioBox` operator formalises a key safety constraint stating that *only one child widget can be selected in a given radio box at any time* [3].

```
isWDRadioBox < predicate >
    (o : Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)) :
    well−definedness  isWDOntology(o)
isWDEditBoxNumeric < predicate >
    (o : Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)) :
    well−definedness  isWDOntology(o)
isWDARINC661Ontology < predicate >
    (o : Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)) :
    ...
```

Listings 8.2: `ARINC661Theory` - basic tester operators

`ARINC661Theory` provides a collection of operators for the construction of a well-defined ARINC 661 ontology, querying and validly modifying the ontology. The list of operators is complete enough to perform basic operations. The `consARINC661Ontology` operator returns an ontology formalising the ARINC 661 widgets: `Ontology(ARINC661Classes, ARINC661Properties, ARINC661-Instances)`. This operator builds a well-defined ontology provided that the

[3] More details are available in Section 3.3.34 page 184 of ARINC 661 standard [18].

arguments passed in satisfy the well-definedness conditions. There is one well-definedness condition, which states that the components related to instances keep the ontology consistent: `isWDOntology` is valid. Particularly, the condition states that the instances are linked according to the class schema: `isWDInstanceAssociations` (see section 5.2).

In Listing 8.3, `CkeckOfSubsetA661OntologyInstances` is a predicate operator verifying that machine variables are compliant with the supplied ontology. This means that the variable is included in the valid set of triples instance, properties and values. The operator guarantees a valid verification provided that the ontology passed in is well-defined.

consARINC661Ontology $< expression >$
$(ii : \mathbb{P}(ARINC661Instances), cii : \mathbb{P}(ARINC661Classes \times ARINC661Instances),$
$ipvs : \mathbb{P}(ARINC661Instances \times ARINC66Properties \times ARINC661Instances)) :$
$Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)$
well−definedness
$isWDARINC661Ontology(consOntology(ARINC661Classes, ARINC66Properties,$
$ii, wellBuiltClassProperties,$
$wellbuiltTypesElements \cup cii, wellBuiltClassAssociations, ipvs))$
CkeckOfSubsetA661OntologyInstances $< predicate >$
$(o : Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances),$
$ui : \mathbb{P}(ARINC661Instances \times ARINC66Properties \times ARINC661Instances)) :$
well−definedness $isWDARINC661Ontology(o)$

Listings 8.3: `ARINC661Theory` - constructor operator

8.3.3 `ARINC661Theory` - Primitives Definitions

The ARINC 661 operators and constants are defined in an axiomatic way. The `AXIOM` section contains the definitions of the several elements declared in Listings 8.1, 8.2, and 8.3.

AXIOMS
$ARINC661ClassesDef :$
$partition(ARINC661Classes, \{Label\}, \{RadioBox\}, \{CheckButton\}, \{PushButton\},$
$\{EditBoxNumeric\}, \{CheckButtonStateClass\}, \ ...)$
$ARINC661PropertiesDef$
$partition(ARINC661Properties, \{hasVisible\}, \{hasEnable\}, \{hasChildrenForRadioBox\},$
$\{hasCheckButtonState\}, \{hasLabelStringForLabel\}, \{hasChildrenForRadioBox\},$
$\{hasCheckButtonState\}, \{hasValue\}, \ ...)$
$ARINC661InstancesDef$
$partition(ARINC661Instances, \{A661_TRUE\}, \{A661_FALSE\}, ..., \{SELECTED\},$
$\{UNSELECTED\})$
$...$
$consARINC661Ontology :$
$\forall ii, cii, ipvs \cdot ii \in \mathbb{P}(ARINC661Instances) \wedge$
$cii \in \mathbb{P}(ARINC661Classes \times ARINC661Instances)$
$\wedge ipvs \in \mathbb{P}(ARINC661Instances \times ARINC661Properties \times ARINC661Instances)$
$\wedge wellbuiltTypesElements \cap cii = \varnothing \wedge ii \subseteq WidgetsInstances$
$\Rightarrow$
$consARINC661Ontology(ii, cii, ipvs) =$
$consOntology(ARINC661Classes, ARINC661Properties, ii,$
$wellBuiltClassProperties,$
$wellbuiltTypesElements \cup cii,$
$wellBuiltClassAssociations,$
$ipvs)$

Listings 8.4: `ARINC661Theory` - axioms for constants and constructor operator

The `ARINC661ClassesDef` axiom defines all the elements of `ARINC661Classes` which is the set of all the widget types and basic type captured from the ARINC 661 standard. For example, `Label` is a widget and `CheckButtonState` is a basic type containing `SELECTED` and `UNSELECTED` values corresponding to the selected and unselected states. In the same vein, ARINC 661 widget attributes are formalised as properties which are grouped in ARINC661Properties —see the `ARINC661PropertiesDef` axiom. THe `ARINC661InstanceDef` axiom populates `ARINC661Instaces` abstract type with instances of all the different widgets like `RadioBoxInstances` and values like `SELECTED`.

In Listing 8.4, the construction axiom `consARINC661Ontology` defines the building of the ARINC 661 ontology using the general construction operator of ontologies imported from `OntologiesTheory`. Note that only components related to instances are required to build the ontology since the classes and properties are extracted once and for all from the ARINC 661 standard specification document.

```
isWDRadioBox
      ∀o · o ∈ Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)
            ⇒ (isWDRadioBox(o) ⇔
      (
         ∀rb, b1, b2 · rb ∈ RadioBoxInstances ∧ b1 ∈ CheckButtonInstances∧
         b2 ∈ CheckButtonInstances∧
            rb ↦ hasChildrenForRadioBox ↦ b1 ∈ getInstanceAssociations(o)∧
            rb ↦  hasChildrenForRadioBox ↦ b2 ∈ getInstanceAssociations(o)
         ⇒ (b1 ↦ hasCheckButtonState ↦ SELECTED ∈ getInstanceAssociations(o)∧
         b2 ↦ hasCheckButtonState ↦ SELECTED ∈ getInstanceAssociations(o)
            ⇒ b1 = b2)
      )
      ∧...
isWDEditBoxNumeric :
   ∀o · o ∈ Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances) ⇒
            (isWDEditBoxNumeric(o) ⇔
            (∀ed, v · ed ↦ hasValue ↦ v ∈ getInstanceAssociations(o) ⇒
               v ∈ A661_EDIT_BOX_NUMERIC_ADMISSIBLE_VALUES))
isWDARINC661Ontology :
   ∀o · o ∈ Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances) ⇒
            (isWDOntology(o) ∧ isWDRadioBox(o) ∧ isWDEditBoxNumeric(o) ⇒
                  isWDARINC661Ontology(o))
CkeckOfSubsetA661OntologyInstance :
   ∀o, ipvs · o ∈ Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)∧
   ipvs ∈ ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances) ⇒
   (isWDARINC661Ontology(consOntology(getClasses(o), getProperties(o), getInstances(o),
      getClassProperties(o), getClassInstances(o), getClassAssociations(o),
      getInstanceAssociations(o)))
                  ⇒ CkeckOfSubsetA661OntologyInstance(o, ipvs))
```

Listings 8.5: `ARINC661Theory` - tester operators

Finally, the predicate operator definitions are specified for formalising verification conditions in an axiomatic way. A significant predicate is that asserting that a well-defined ARINC 661 domain ontology is, first of all, a well-defined ontology in the sense of `isWDOntology`, and then other conditions are appended like that formalised in `isWDRadioBox` predicate holds —see the axiom `consARINC661OntologyDef` in Listing 8.5. Indeed, a key condition underlying all the operators is formalised in the predicate `isWDARINC661Ontology`, which requires that the ontology passed in must be a well-defined ontology, formalises all the constraints of the ARINC 661 standard. Indeed it gathers all the con-

ditions of the widgets like `isWDARadioBox` and `isWDEditBoxNumeric`. The last predicate states that the entered value must be within a specified interval. Last, an important axiom is `ckeckOfSubsetA661OntologyInstance` which axiomatises the verification that a collection of triples (instance, property, value) represented by the argument `ipvs` are coherent with the constraints of the given ontology `o`.

8.3.4 `ARINC661Theory` - Theorems

The `THEOREM` section in Listing 8.6 contains important facts deduced from the axiomatisation of the constants and operators. The well-definedness of the ontology produced by the construction operator `consARINC661Ontology` is ensured by theorems `isWDClassProperitesThm` and `isWDClassAssociationsThm`. They describe two important facts: classes are related to properties which do exist in the ontology (`isWDClassProperitesThm`) and class associations connect classes and properties which are in the `classes` and `properties` components of the ontology (`isWDClassAssociationsThm`) —see section 5.2 for their formal definitions.

The proof relied on the well-definedness conditions of the construction operator and its direct definition which, when combined, entail the well-definedness of the two components of the constructed ontology. Specifically, the component involved in the definition of `consARINC661Ontology` (see Listing 8.4), namely `wellBuiltClassProperties` and `wellBuiltClassAssociations` are defined so that the well-definedness conditions are entailed.

```
THEOREMS
isWDClassProperitesThm
   ∀ii, cii, ipvs · ii ∈ ℙ(ARINC661Instances)
   ∧cii ∈ ℙ(ARINC661Classes × ARINC661Instances)
   ∧ipvs ∈ ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances)
   ∧wellbuiltTypesElements ∩ cii = ∅ ∧ ii ⊆ WidgetsInstances
                  ⇒
                  isWDClassProperites(consARINC661Ontology(ii, cii, ipvs))
isWDClassAssociationsThm
   ∀ii, cii, ipvs · ii ∈ ℙ(ARINC661Instances)
   ∧cii ∈ ℙ(ARINC661Classes × ARINC661Instances)
   ∧ipvs ∈ ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances)
   ∧wellbuiltTypesElements ∩ cii = ∅ ∧ ii ⊆ WidgetsInstances ⇒
                  isWDClassAssociations(consARINC661Ontology(ii, cii, ipvs))
END
```

Listings 8.6: `ARINC661Theory` - theorems

8.4 Weather Radar Application Case Study

This section [4] is dedicated to modelling the user interface of the Weather Radar System (WXR) of the Multi-Purpose Interactive Application. The description of the case study is given in Section 4.2. This section corresponds to phase (3) on Figure 8.1. The theory is composed of two axiomatic blocks; the first block

[4]The full Event-B development of WXR case study is in appendix in C.4

`WXRUIDescriptoinAxiomatisaiton` defines widgets of the user interface and the second block `EventsAffectingWidgetsAxiomatisation` defines operators for formalising the interaction with the user interface. Subsection 8.4.2 is related to the instantiation of the `ARINC661Theory` where the instances specific to the user interface of the WXR application are defined and provided to correctly complete the ontology. Subsection 8.4.3 presents the operator used for updating the user interface of the WXR application. Last, subsection 8.4.4 discusses key theorems formalising important facts about the formal model of the user interface.

8.4.1 `WXRTheory` - Instances Declaration

```
THEORY
    WXRTheory
  IMPORT THEORY
    ARINC661Theory
  AXIOMATIC DEFINITIONS
    WXRUIDescriptoinAxiomatisaiton :
    OPERATORS
      A661WXROntology < expression > () :
        Ontology(ARINC661Classes, ARINC661Properties, ARINC661Instances)
      Instances < expression > () : ℙ(ARINC661Instances)
      ClassInstances < expression > () : ℙ(ARINC661Classes × ARINC661Instances)
      MODESELECTIONLabel < expression > () : ARINC661Instances
      OFF1Label < expression > () : ARINC661Instances
      OFF1CheckButton < expression > () : ARINC661Instances
      ...
      WXRFeatures < expression >
        (o : Ontology(ARINC661Classes, ARINC661Properties, ARINC661Instances)) :
            ℙ(ARINC661Instances × ARINC661Properties × ARINC661Instances)
        well−definedness
            isWDARINC661Ontology(o)
```

Listings 8.7: `WXRTheory` - constants

`WXRTheory` includes constants and operators dealing with instance information, which is not defined in `ARINC661Theory`. This theory provides elements to formally describe WXR the user interface. `WXRFeature` contains the instances association used by the WXR model —a set of triples (instance, property, value) of type $\mathbb{P}(ARINC661Instances \times ARINC661Properties \times ARINC661Instances)$. In Listing 8.7, several constants are declared such as `OFF1Label` which represents the label OFF in the model selection radio box (see Figure 8.2). In the same manner, `OFF1CheckButton` represent the OFF check box. In the same vein, other widgets are declared in this part of `WXRTheory`.

8.4.2 `WXRTheory` - Instances Definition

In Listing 8.8, instances of the ARINC 661 ontology theory are defined as constants of the type $\mathbb{P}$ (`ARINC661`). For example, `WXRinstances` is a set of all possible widgets of the user interface: `WXRLabels`, `WXRCheckButtons`, etc. The `WXRFeatures` operator restricts the ARINC661 ontology to the instances needed to design the WXR user interface, i.e. *none of these instances is outside of the ARINC 661 ontology.* `Instances` is composed of several blocks: `WXRLabels`,

`WXRcheckButtons`, `WXREditNumericBoxes`, ... etc. Then, the instances are affected by relevant classes in `ClassInstancesDef` axiom. The instance associations are also defined in `WXRInstancePropertyValuesDef`. Last, these components are used for defining the construction of ontology as shown in axiom `A661WXROntology`.

```
AXIOMS
  WXRLabelsDef :
   partition(WXRLabels, {MODESELECTIONLabel}, {OFF1Label}, ...)
  WXRcheckButtons :
    partition(WXRcheckButtons, {OFF1CheckButton}, {STDBYCheckButton},
    {TSTCheckButton}, {WXONCheckButton}, {WXACheckButton})
  WXRradioBoxesDef :
    partition(WXRradioBoxes, {WXRradioBoxModeSelection},
    {WXRradioBoxTiltSelection}, {WXRradioBoxStabilization})
  WXREditNumericBoxesDef :
    WXREditNumericBoxes = {TITLANGLEEditNumericBox}
  WXRInstancesDef :
    partition(Instances, WXRLabels, WXRcheckButtons, WXRStrings,
              WXRToggleButtons, WXREditNumericBoxes,
              WXRradioBoxes, WXRWidgetIdents)
  WXRClassInstancesDef :
    ClassInstances = ({Label} × WXRLabels)∪
                ({ARINC661_STRING_CLASS} × WXRStrings)∪
                ({CheckButton} × WXRcheckButtons)∪
                ({ToggleButton} × WXRToggleButtons)∪
                ({RadioBox} × WXRradioBoxes)∪
                ({WidgetIdentClass} × WXRWidgetIdents)
     ...
  WXRInstancePropertyValuesDef :
    InstanceAssociation = iaLabels ∪ iaCheckBttons∪
                                iaToggleButtons ∪ ioRadioBoxes
  A661WXROntologyDef :
    A661WXROntology = consARINC661Ontology(Instances,
                                ClassInstances, InstanceAssociation)
  WXRFeaturesDef :
      ∀o · o ∈ Ontology(ARINC661Classes,
           ARINC66Properties,
           ARINC661Instances)∧
      isWDARINC661Ontology(o) ⇒
          WXRFeatures(o) = InstanceAssociation
```

Listings 8.8: `WXRTheory` - axioms for constants

8.4.3 `WXRTheory` - Operators Declaration and Definition

```
AXIOMATIC DEFINITIONS    EventsAffectingWidgetsAxiomatisation :
    OPERATORS
      isWDChangeModeSelection < predicate >
          (o : Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances),
          ui : ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances),
          mode : ARINC661Instances) :

      changeModeSelection < expression >
          (o : Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances),
          ui : ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances),
          mode : ARINC661Instances)
          : ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances)
        well−definedness  isWDChangeModeSelection(o, ui, mode)
     ...
```

Listings 8.9: `WXRTheory` - operators declarations

The user interface provides user interaction operators: choosing a mode selection, switching between the two states of the stabilization and tilt section feature and finally input a new tilt angle value. Each interaction is modelled by two operators: a WD predicate and an interactions modelling operator. For example, `isWDChangeModeSelection` and `changeModeSelection` pair of operators deal with mode selection change (see Listing 8.9). Other operators are defined and associated with well-definedness conditions to allow for interaction with a user interface which is depicted in Figure 8.2.

In the `AXIOMS` clause, several operators are defined (see Listing 8.10). For example, `changeModeSelection` is associated with a well-definedness condition formalised by the predicate operator `isWDChangeModeSelection`. This operator states that *crew members may select only specified modes in* `WXRcheckButtons` and `CkeckOfSubsetA661OntologyInstances` (see Listing 8.5) ensures that the `ui` parameter complies with ontology rules and constraints. `changeModeSelection` operator allows to interact with the interface passed in as an argument, and it permits to update to the mode of functioning of the radar. When the conditions including that related to the ontology `o` (`isWDOntology`) formalised by `isWDChangeModeSelection` hold, the `changeModeSelection` updates the `ui` argument specifying the user interface by deleting the triple (instance, property, value) $mode \mapsto hasCheckButtonState \mapsto UNSELECTED$ and by adding the triple $mode \mapsto hasCheckButtonState \mapsto SELECTED$.

AXIOMS

$$
\begin{array}{l}
isWDChangeModeSelectionDef: \\
\quad \forall o, ui, mode \cdot o \in Ontology(ARINC661Classes, ARINC66Properties, \\
\quad ARINC661Instances) \land \\
\quad ui \in \mathbb{P}(ARINC661Instances \times ARINC66Properties \times ARINC661Instances) \land \\
\quad mode \in ARINC661Instances \Rightarrow \\
\quad (isWDChangeModeSelection(o, ui, mode) \Leftrightarrow \\
\quad CkeckOfSubsetA661OntologyInstances(o, ui) \land \\
\quad mode \in WXRcheckButtons) \\
changeModeSelectionDef: \\
\quad \forall o, ui, mode \cdot o \in Ontology(ARINC661Classes, ARINC66Properties, \\
\quad ARINC661Instances) \land \\
\quad ui \in \mathbb{P}(ARINC661Instances \times ARINC66Properties \times ARINC661Instances) \land \\
\quad mode \in ARINC661Instances \Rightarrow \\
\quad (changeModeSelection(o, ui, mode) = \\
\quad (ui \setminus \{i \mapsto hasCheckButtonState \mapsto UNSELECTED \mid \\
\quad i \mapsto hasCheckButtonState \mapsto SELECTED \in ui \land \\
\quad i \in (WXRcheckButtons \setminus mode)\}) \cup \\
\quad mode \mapsto hasCheckButtonState \mapsto SELECTED) \\
\ldots
\end{array}
$$

Listings 8.10: `WXRTheory` - operator definitions

8.4.4 `WXRTheory` - Theorems

In `WXRTheory`, important safety properties are stated and proved in the form of theorems. Two key facts are formalised and proved for all operators; the suffixes `CkeckOfSubsetA661OntologyInstancesInst` and `Safely` is used to denote each of them.

First, `Safely`-suffixed theorems assert that the operators guarantee that the exclusive section of the check button is preserved in all radio boxes of the user

interface and that the values of the edit boxes never overflow. The second class of theorems ensures that the instances of modelling the user interface always comply with the ontology constraints.

```
THEOREMS
isWDARINC661Ontology :
  isWDARINC661Ontology(A661WXROntology)
WXRFeaturesSafely :
    ∀o, ipvs · isVariableOfARINC661Ontology(o, ipvs)∧
    (ipvs = WXRFeatures(o))
      ⇒ ...
WXRFeaturesCkeckOfSubsetA661OntologyInstances :
    ∀o, ipvs · isWDARINC661Ontology(o)∧
    ipvs ∈ ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances)∧
    (ipvs = WXRFeatures(o)) ⇒ CkeckOfSubsetA661OntologyInstances(o, ipvs)
changeModeSelectionSafety :
    ∀o, ipvs · CkeckOfSubsetA661OntologyInstances(o, ipvs)∧
      (∃uiArg·
          (∃m · isWDChangeModeSelection(A661WXROntology, uiArg, m)∧
          ipvs = changeModeSelection(A661WXROntology, uiArg, m)))
      ⇒ (
              ∀rb, b1, b2 · rb ∈ RadioBoxInstances ∧ b1 ∈ CheckButtonInstances∧
              b2 ∈ CheckButtonInstances∧
              rb ↦ hasChildrenForRadioBox ↦ b1 ∈ ipvs∧
              rb ↦ hasChildrenForRadioBox ↦ b2 ∈ ipvs ⇒
              (b1 ↦ hasCheckButtonState ↦ SELECTED ∈ ipvs∧
              b2 ↦ hasCheckButtonState ↦ SELECTED ∈ ipvs ⇒ b1 = b2)) ∧ ...
changeModeSelectionIsCkeckOfSubsetA661OntologyInstances :
    ∀o, ipvs · isWDARINC661Ontology(o)∧
    (∃uiArg · (∃m· isWDChangeModeSelection(A661WXROntology, uiArg, m) ∧ ipvs =
    hangeModeSelection(A661WXROntology, uiArg, m)))
          ⇒ CkeckOfSubsetA661OntologyInstances(o, ipvs)
...
END
```

Listings 8.11: `WXRTheory` - theorems

`changeModeSelectionIsCkeckOfSubsetA661OntologyInstances` is a theorem stating that the operator `changeModeSelection` preserves the compliance between the user interface model (`ui` argument) and the ontology of ARINC 661 (the argument `o`). `changeModeSelectionSafety` states that the operator preserves the properties of exclusive selection in a radio box and that the input values in edit boxes always belong to the allowed values. In Listing 8.11, the two facts are illustrated for the initialisation operator `WXRFeatures`.

8.4.5 Annotated Model of WXR —(4) on Figure 8.1

Figure 8.2 depicts the composition, in terms of widgets, of the user interface of the WXR application. The formal elements in rectangles are beside the widgets to illustrate the correspondence between widgets and their formal counterparts.

The Weather Radar Application user interface is modelled as an Event-B machine which uses elements defined in `WXRTheory`. In Listing 8.12, the state of the user interface is modelled by `uiStateVar` variable representing the state of this user interface, it may evolve when the events formalising the interactions of the user are triggered. The event `changeModeSelectionEvt` models the interaction on the mode selection radio box where only one check box shall be selected at once. Similarly, other events are defined to represent other interactions.

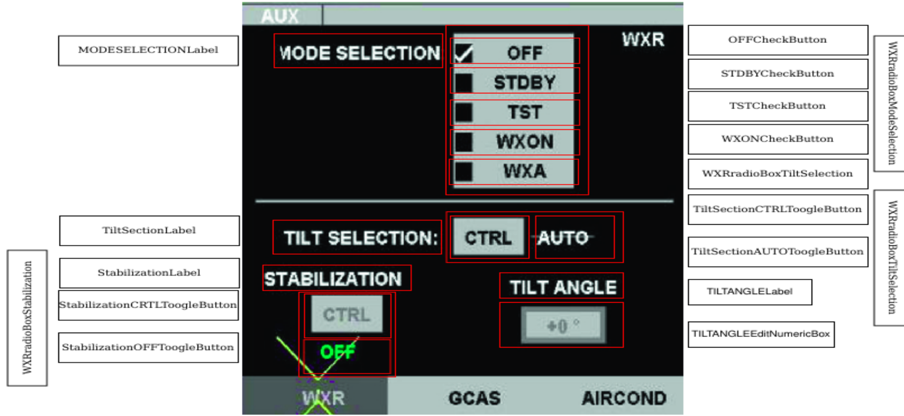


Figure 8.2: WXR system annotated with ARINC 661 concepts

```
MACHINE WXRModel
VARIABLES   uiStateVar
INVARIANTS
  TypingAndClosedness :
    ∃uiArg · ((uiStateVar = initiator(A661WXROntology))∨
           ∃m· isWDChangeModeSelection(A661WXROntology, uiArg, m)∧
               uiStateVar = changeModeSelection(A661WXROntology, uiArg, m))∨
           (∃v· isWDChangeTitlAngle(A661WXROntology, uiArg, v)
              ∧uiStateVar = changeTitlAngle(A661WXROntology, uiArg, v))∨
           (isWDSetTiltSelectionCTRL(A661WXROntology, uiArg)∧
              uiStateVar = setTiltSelectionCTRL(A661WXROntology, uiArg))∨
           (isWDSetTiltSelectionAUTO(A661WXROntology, uiArg)∧
              uiStateVar = setTiltSelectionAUTO(A661WXROntology, uiArg))∨
           (isWDSetStabilizationCTRL(A661WXROntology, uiArg)∧
              uiStateVar = setStabilizationCTRL(A661WXROntology, uiArg))∨
           (isWDSetStabilizationOFF(A661WXROntology, uiArg)∧
              uiStateVar = setStabilizationOFF(A661WXROntology, uiArg))
  OntologyComplianceThm :
           CkeckOfSubsetA661OntologyInstancesIns(A661WXROntology, uiStateVar)
  ARINC661SafetyThm :
           (∀rb, b1, b2 · rb ∈ RadioBoxInstances∧
           b1 ∈ CheckButtonInstances ∧ b2 ∈ CheckButtonInstances∧
           rb ↦ hasChildrenForRadioBox ↦ b1 ∈ uiStateVar∧
           rb ↦ hasChildrenForRadioBox ↦ b2 ∈ uiStateVar
              ⇒ (b1 ↦ hasCheckButtonState ↦ SELECTED ∈ uiStateVar∧
              b2 ↦ hasCheckButtonState ↦ SELECTED ∈ uiStateVar ⇒ b1 = b2)) ∧ ...
EVENTS
  INITIALISATION
  THEN
    act1 : uiStateVar := WXRFeatures(A661WXROntology)
  END
  changeModeSelectionEvt
  ANY   mode
  WHERE
    grd1 : mode ∈ WXRcheckButtons
    grd2 : isWDChangeModeSelection(A661WXROntology, uiStateVar, mode)
  THEN
    act1 : uiStateVar := changeModeSelection(A661WXROntology, uiStateVar, mode)
  END
END
```

Listings 8.12: Event-B machine for modelling WXR user interface

The domain-related properties are proved deductively and the theorems of the ARINC 661 theory are transferred to the model. This is true provided, as prescribed by the framework, only the operators supplied by the theory are used to define the events. In Listing 8.12, the invariant `TypingAndClosedness` formalises this methodological requirement, stipulating that only the operator provided by the theory is allowed to manipulate the variables. Moreover, it permits the annotation of the `uiStateVar` through typing (Event-B theories support type inference)

Indeed, the desired safety properties of the systems are expressed as theorems, `OntologyComplianceThm` and `ARINC661SafetyThm`. The first establishes that the instances associations comply with the definition of the ARINC 661 ontology. The second ensures that the safety properties of ARINC 661 widgets such as the selection of check buttons of the same radio box are exclusive, and the values of an edit box never overflow.

Furthermore, the `changeModeSelectionEvt` event uses the `changeModeSelection` operator to select a mode from the mode selection radio box, such as *STDBY* (see Figure 8.2). Note that this event is guarded with the `isWDChangeModeSelection` operator which corresponds to its well-definedness condition imported from `WXRTheory`.

8.5 Advantages of The Framework

This section discusses the advantages of our approach for standard conformance-by-construction. This assessment is drawn from the formalisation of the industrial standard ARINC 661 and the modelling of the case study presented in sections 8.1 and 8.3. The framework has four main benefits compared to ad hoc and implicit integration of the standards rules and criteria directly in the formal model. These advantages are discussed below.

8.5.1 Achieving Standard Conformance Formally

The functional goal of the framework has been achieved. The framework demonstrated that standard conformance may be checked formally. Indeed, domain-related requirements specified by an industrial standard like ARINC 661 may be transferred and enforced properly on formal design models. Specifically, the requirement of selecting exclusively one radio box at once has been proved to hold on the Weather Radar System user interface by formally deducing it. In contrast, conventionally requirement would be formalised as an invariant therefore it would necessitate proving the property as safety property inductively, i.e. each event preserves the property and the initialisation event starts the induction.

8.5.2 Qualitatively Enhanced System Models

The WXR model has been greatly improved as a result of extensive outsourcing of safety properties to the theory level and the use of the ontology description theory. The use of a theory validated by experts led to trustworthy models, which may be considered as the clean organisation of formal specification. In addition, this approach enabled domain-specific (standards) models to be validated, once and for all, independently of the systems design models. The resulting models realised a qualitative leap compared to the ad hoc and implicit integration of this domain knowledge inside the formal models. The framework features the separation of concerns and permits the extension and reusability of the generic formal theories.

8.5.3 Reduction of Modelling and Proving Effort

Although the description of the domain-specific theory, `ARINC661Theory`, requires a significant amount of modelling effort, the specification of the models is simplified as a result of the existence of such theory. Indeed, it provides operators which are used as high-level abstraction building blocks. The models are composed of these operators, thus, transferring the desired safety properties provided that well-definedness conditions are supplied. The transferring process is valid since properties have been proved once and for all on the theory side, and the model is built exclusively using the theory primitives. In this way, the modelling effort is better reused and shared leading to a capitalised modelling effort in the long run.

The proving process is streamlined. Indeed, the well-definedness proof obligations are discharged semi-automatically thanks to well-definedness predicates which are, by principle, provided in the domain theory. Moreover, `INV` proof obligations are discharged automatically because the working hypothesis requires that the model shall be built using only the primitives of the theory, including the operators. In addition, the theorems in the machine (see Listing 8.12) are proved through a single *modus ponens* inference rule using the theorems of the theory.

Event-B Models and Theories	**Number of proof obligations**
OntologiesTheories	21
ARINC661Theory	10
WXRTheory	39
WXRModel	18

Table 8.2: Proof statistics of conformance checking and MPIA case study

Table 8.2 shows 88 automatically generated proof obligations for the theories and `WXRModel` Event-B machine. Only 18 out of the 88 proof obligations in total are generated for the Event-B system model (machine) and automatically discharged. In particular, the next models of intercative critical systems using

the framework and the Event-B theory `ARINC661Theory` will only need to prove the proof obligations of the system model; without the need to prove again the proof obligations of the theory.

8.5.4 Enabling Evolution of Standard

Last but not least, the approach enables the evolution of the standard. Indeed, the neat separation of the common domain knowledge from system specifics fosters the separation of concerns principle and orthogonality of the evolution principle. Both domain models and system design models may evolve asynchronously or independently with limited impact on each other. From a proof perspective, only proof obligations caused by the evolution need to be discharged.

8.6 Conclusion

This section concludes the chapter dedicated to formal conformance. This chapter presented the application of the general methodology of integrating domain knowledge in formal models discussed in Chapter 5 and Chapter 6 to checking conformance of systems to standards. For this purpose, section 8.1 addressed the issue of standard formal conformance where a formal framework for achieving formal conformance regarding a standard is presented. The ARINC 661 standard and user interface of the Weather Radar System critical system were used for demonstrating the effectiveness of the framework. Section 8.3 discussed the formalisation of the ARINC 661 standard specification dedicated to aircraft Cockpit Display System (CDS) and section 8.4 was dedicated to the description of the WXR user interface case study.

From the standardisation point of view, industry consortia and standardisation bodies may define, through consensual agreement, public formal theories modelling domain standards which may significantly increase the reliability of the certification process.

Conclusion and Perspectives

Conclusion

Formal methods demonstrated their effectiveness in improving confidence in critical system modelling by revealing unsafe design decisions, therefore preventing them from reaching the system implementation. This thesis is part of a series of works intended to further enhance formal modelling by making explicit modelling of domain knowledge in formal modelling and verification. Although mathematical knowledge such as algebraic structures, real numbers, differential equations and so on have been largely addressed across several formal methods, the challenge of systematically handling explicit modelling of domain knowledge remains a challenge. In this thesis, we, first, addressed the challenge of explicit modelling of domain knowledge, and identified potential improvements of explicit modelling of domain knowledge over classical modelling which adopts an *ad hoc* and implicit modelling of domain requirements. This ad hoc modelling of domain yields to monolithic specification of domain-specific along system-specific requirements. Next, we emphasised the relevance of domain modelling for interactive critical systems to illustrate our framework on concrete systems. Then, we stated desired features for the framework supporting explicit modelling of domain knowledge and domain-specific properties transferring to system models. Among these features, the ontology formalism was preferred for representing domain knowledge for uniformity and consensuality reasons. Consequently, domain requirements such as safety properties and behavioural analyses have been considered, and conformance checking of system model regarding standards has been addressed. Last, the framework has been implemented in Event-B method by leveraging the use of generic Event-B theories. The work carried out during this has resulted in the following contributions:

Ontology-Based Explicit Modelling of Domain Knowledge. We developed a framework for explicit modelling of domain knowledge in formal modelling and properties transferring to such system models. A contrasting comparison is unrolled through the parallel development of two versions of a didactic model of temperature aggregator and TCAS case study. The first versions has been de-

veloped following an ad hoc handling of the domain knowledge, and the second version has been developed using our framework by modelling explicitly domain knowledge requirements. As a result, the improvement of the quality of models has been demonstrated in terms of modular design, separation of concerns, enhanced understandability, and the reductions of modelling and proof effort as well as the reuse and sharing of domain theories. Furthermore, structuring formal models such that they reference explicitly domain knowledge offers an orthogonal and independent evolution of domain and system specifications. Next, we formalised an ontology modelling language for describing engineering domains allowing to design a uniform and homogeneous collection of domain ontologies. This ontology modelling language has been implemented as a generic Event-B theory composed of a unique data type representing concepts, properties, instances and associations of the intended ontology, and a set of operators, axioms and theorems. Last, several ontologies have been defined to represent various domains using this ontology modelling language.

Annotation-Based Domain Safety Properties Transferring. Starting from analysis of requirements on interactive systems case studies, provided that the domain knowledge is explicitly modelled as a separate specification, we showed that it is possible to transfer the safety requirements entailed by the domain specification to system models. Formally, it means that the theorems of the domain theory are also theorems of the system model. Then, we emphasised the role of *working hypothesis of our framework* requiring that the system model shall be specified using the primitives of the domain theories *exclusively* in order to transfer correctly the domain properties. Our framework has been illustrated on the TCAS case study through the safety property requiring that critical aircraft should be always visible.

Annotation-Based Domain-Specific Behavioural Analysis. We designed a methodology to define domain-specific behavioural analyses based on annotation. Then, these analyses may be applied to investigate the behaviours of state-based Event-B models, and check behavioural requirements specified in engineering domain knowledge. The methodology relies on our ontology modelling language for describing ontologies of events and EB4EB [143] reflexive Event-B framework to define the behavioural analyses. An annotation-based mechanism has been specified to link domain concepts and requirements to system model. Therefore, it enables to check not only the plain temporal behavioural of the model but also to verify behaviours of the models given an ontology (i.e. hierarchy) of the model events. This methodology is non-intrusive in the sense that it may be used at any stage of the refinement chain of the development. Last, this approach was illustrated on a model of an ATM that has been investigated to verify that input passcodes are always followed by confirmation or abortion. Moreover, the last analysis is an instantiation of the analysis allowing to check whether an event annotated by a source ontology concepts (i.e. input tag) is eventually followed by events annotated by target ontology concepts meanwhile a specific category of events are exclusively allowed to be triggered.

Conformance Checking With Respect to Standard Requirements. The idea of conformance checking to a standard started with the observation that standard specification may be regarded as a special domain knowledge requirements. Our framework has been used to show that standard specification and conformance checking may be regarded as a special case of explicit modelling of domain knowledge. Indeed, we proposed a methodology for formalising a standard specification as a knowledge model with an ontology, and check that a system model is compliant with the requirements specified by the standard by transferring the properties entailed by the standard specification. Our proposal has been exemplified by (1) specifying the part of the section of the ARINC 661 standard dedicated to widgets as a domain ontology, (2) modelling the WXR user interface by referring to the widget primitives of the ARINC 661 Event-B theory. As a consequence, the formalised ARINC 661 standard properties of the widgets (such as the selection of buttons of a radio box shall be exclusive) are transferred to the system model; they are proved to be theorems of the model. The conformance is achieved thanks to the transfer of the properties from the formalisation of the standards requirements to the system models.

Perspectives

A number of extension directions have been identified to improve the current state of the framework. They are discussed below.

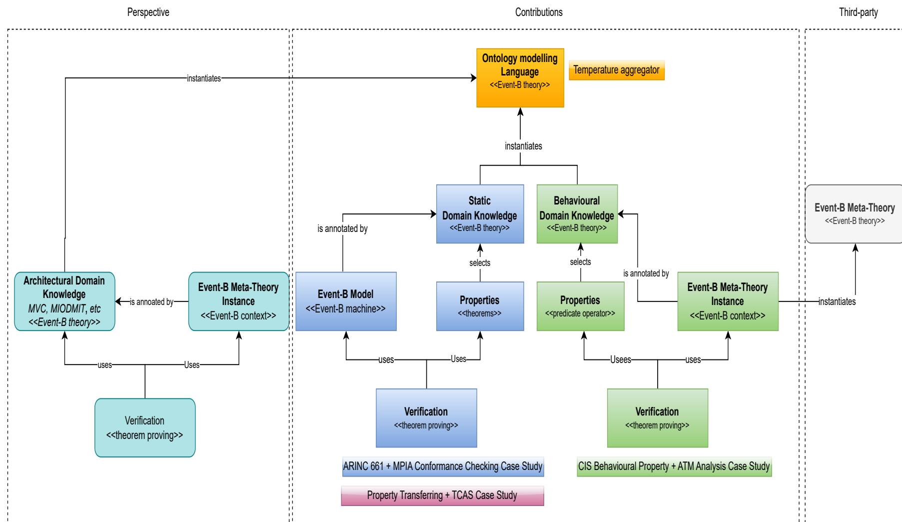


Figure 8.3: A map of thesis contributions and perspective

In Figure 8.3, that the part coloured in aqua shows a possible extension to the map of contributions to include architectural domain knowledge (i.e. *Perspectives* box). Besides the figure contains the contributions related to static

domain knowledge (safety requirements) and behavioural domain knowledge (domain-specific analyses) encircled in the *Contributions* box. The *Third party* box represents the Event-B meta-theory.

Design Pattern as Domain-Specific Structural Analysis. In this thesis, we proposed a framework for explicit modelling of domain knowledge, and we formalised it using the Event-B method. A different type of domain knowledge is the architectural requirements to which a system model need to comply with [141]. An annotation-based approach may be effective, where the architectural requirements should be formalised as a domain theory, and the Event-B system model should be exported to the EB4EB framework. Last, we can use the Event-B proving mechanism to establish the compliance.

Extended Formalisation of Standards. The methodology devised for conformance checking of system model regarding standard specifications may be used to address the required safety assurances to meet certification standards. The formally proved properties and the formal data types and operators can be used as an evidence in assurance cases, helping the certification process by guiding both the development and regulatory evaluation of interactive critical systems. Last, from the standardisation point of view, industry consortia and standardisation bodies may define formal processes addressing consensual agreement on the definition and consistence of the formal theories modelling domain standards i.e. the process consisting in analysing text-based standards in order to derive domain standard theories and in validating these derived theories.

In addition, this work can be completed by the study of other type of domain standards related to temporal properties, real-time scheduling, common criteria for security etc. and application domains like avionics, transportation systems and so on. In the same vein, we believe that the approach of defining domain-specific behavioural analyses can be exploited for certification purposes. Indeed, a non-intrusive analysis may be carried out for such purposes if certification standards are formalised as theories formalising certification properties.

Richer Ontology of Events. In the methodology for defining domain-specific behavioural analyses, we illustrated a domain composed of a hierarchy of concepts. As future work, we plan to apply the framework to other case studies and generalise the ontology of events to include associations between ontology concepts.

Ontology Hierarchical Behavioural Operators. In this thesis, domain theories presented a flat collection of operators. However, collections of operators hierarchically organised should allow to describe a domain model at different levels of abstraction. Therefore, special behavioural system models may be derived from a general domain enjoying a desired safety property entailed by the domain theory. The refinement provided by Even-B may be useful to ensure that the latter model using the special operators preserves the properties of the former model described using the general operators. This extension of the framework

would permit to prove only once the preservation of the safety property by (1) proving that the general operators preserves the property, and (2) that the special operators are derived from the general operators.

A Library of Domain Ontologies. The framework proposed in this thesis may be used to develop domain theories pertaining to different engineering contexts such as autonomous vehicles, railways systems, etc. In general any domain knowledge that can be formalised as an ontology may be addressed by our framework. As a result, when the number of domain theories is large enough, it makes sense to create a library where domain theories can be contributed and reused rather than reinventing current ones. Such a library would enable more complete descriptions of domains provided that a larger number of developers and experts contribute to common projects.

Modularisation of Domain Theories. This perspective is a sequel to the availability of domain theory repository. Indeed, it would be interesting to investigate Modularisation mechanisms of domain theories to study their effects on the preservation of safety properties entailed by the individual domain theories. Examples of modularisation operations are hierarchy, abstraction and composition.

Systematic Assessment of Orthogonal Evolution of Domain and System Specifications. In this thesis, specifically in the contributions related to formal conformance checking, we identified the advantage of orthogonal evolution of domain and system specifications. Indeed, it is clear that if a theory is modified such that the operators keep the extended theory's theorems provable, the system models built using the data types and operators of the extending theory always entail the safety properties of the theory. Formally, the theorems of the extended theory are equally theorems of the extending theory. Therefore, since the models exclusively use the primitives of the extending theory (working hypothesis), then the systems models preserve the safety properties. A systematic investigation and evaluation of both the evolution of theories and systems models on sufficient number of theories and systems models may reveal useful patterns and general conditions where this preservation is always ensured. For the Event-B method, this means that in case of evolution of system models and/or domain knowledge, it will be helpful for the designer to check the only PO's that result from this evolution. Formalising the evolution mechanism so that such PO's are automatically generated, which save development time.

Bibliography

[1] I. 9646. ISO/IEC 9646-1:1994 - Information technology — Open Systems Interconnection — Conformance testing methodology and framework — Part 1: General concepts, 1994. URL: https://www.iso.org/standard/17473.html.

[2] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009. doi:10.1007/s00778-008-0125-y.

[3] J. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010. doi:10.1017/CBO9781139195881.

[4] J. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, 2010. doi:10.1007/s10009-010-0145-y.

[5] J. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Informaticae*, 77(1-2):1–28, 2007. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi77-1-2-02.

[6] J. Abrial and L. Mussat. On Using Conditional Definitions in Formal Theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*, volume 2272 of *Lecture Notes in Computer Science*, pages 242–269. Springer, 2002. doi:10.1007/3-540-45648-1_13.

[7] J.-R. Abrial, M. Butler, S. Hallerstede, M. Leuschel, M. Schmalz, and L. Voisin. Proposals for Mathematical Extensions for Event-B. Technical report, 2009. URL: https://web-archive.southampton.ac.uk/deploy-eprints.ecs.soton.ac.uk/216/.

[8] E. A. S. Agency. Certification Specifications and Acceptable Means of Compliance for Large Aeroplanes CS-25. Standard, 2015. URL: https://www.easa.europa.eu/en/document-library/certification-specifications/cs-25-amendment-25.

[9] Y. Ait Ameur, I. Ait Sadoune, K. Hacid, and L. Mohand Oussaid. Formal Modelling of Ontologies : An Event-B based Approach Using the Rodin Platform. In R. Laleau, D. Méry, S. Nakajima, and E. Troubitsyna, editors, *Proceedings Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX) and Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD), IMPEX/FM&MDD 2017, Xi'an, China, 16th November 2017*, volume 271 of *EPTCS*, pages 24–33, 2017. doi:10.4204/EPTCS.271.2.

[10] Y. Ait Ameur and M. Baron. Formal and experimental validation approaches in HCI systems design based on a shared event B model. *Int. J. Softw. Tools Technol. Transf.*, 8(6):547–563, 2006. doi:10.1007/s10009-006-0008-8.

[11] Y. Ait Ameur, M. Baron, L. Bellatreche, S. Jean, and E. Sardet. Ontologies in engineering: the OntoDB/OntoQL platform. *Soft Comput.*, 21(2):369–389, 2017. doi:10.1007/s00500-015-1633-5.

[12] Y. Ait Ameur, N. Belaid, M. Bennis, O. Corby, R. Dieng-Kuntz, J. Doucy, P. Durville, C. Fankam, F. Gandon, A. Giboin, P. Giroux, S. Grataloup, B. Grilhères, F. Husson, S. Jean, J. Langlois, P. Luong, L. S. Mastella, O. Morel, M. Perrin, G. Pierra, J. Rainaud, I. Ait Sadoune, E. Sardet, F. Tertre, and J. F. Valiati. Semantic Hubs for Geological Projects. In K. Belhajjame, M. d'Aquin, P. Haase, and P. Missier, editors, *First International Workshop on Semantic Metadata Management and Applications, SeMMA 2008, Located at the Fifth European Semantic Web Conference (ESWC 2008), Tenerife, Spain, June 2nd, 2008. Proceedings*, volume 346 of *CEUR Workshop Proceedings*, pages 3–17. CEUR-WS.org, 2008. URL: https://ceur-ws.org/Vol-346/1.pdf.

[13] Y. Ait Ameur, G. Dupont, I. Mendil, D. Méry, M. Pantel, P. Rivière, and N. K. Singh. Empowering the Event-B Method Using External Theories. In M. H. ter Beek and R. Monahan, editors, *Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings*, volume 13274 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2022. doi:10.1007/978-3-031-07727-2_2.

[14] Y. Ait Ameur and D. Méry. Making explicit domain knowledge in formal system development. *Sci. Comput. Program.*, 121:100–127, 2016. doi:10.1016/j.scico.2015.12.004.

[15] Y. Ait-Ameur, S. Nakajima, and D. Méry. *Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems*. Springer Singapore, 2021. URL: https://hal.inria.fr/hal-02910199, doi:10.1007/978-981-15-5054-6.

[16] I. Ait Sadoune and L. Mohand Oussaid. Building Formal Semantic Domain Model: An Event-B Based Approach. In K. Schewe and N. K. Singh, editors, *Model and Data Engineering - 9th International Conference, MEDI 2019, Toulouse, France, October 28-31, 2019, Proceedings*, volume 11815 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2019. doi:10.1007/978-3-030-32065-2_10.

[17] S. Albukhitan, T. Helmy, and M. Al-Mulhem. Semantic Annotation Tool for Annotating Arabic Web Documents. In E. M. Shakshuki and A. Yasar, editors, *Proceedings of the 5th International Conference on Ambient Systems, Networks and Technologies (ANT 2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014), Hasselt, Belgium, June 2-5, 2014*, volume 32 of *Procedia Computer Science*, pages 429–436. Elsevier, 2014. doi:10.1016/j.procs.2014.05.444.

[18] ARINC. ARINC 661 specification: Cockpit Display System Interfaces to User Systems, Prepared by AEEC, Published by SAE, Melford Blvd., Bowie, Maryland, USA, 06 2019. URL: https://www.sae.org/standards/content/arinc661p1-7/.

[19] H. Arnaud, P. A. Palanque, J. L. Silva, Y. Deleris, and E. Barboni. Formal description of multi-touch interactions. In P. Forbrig, P. Dewan, M. Harrison, and K. Luyten, editors, *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'13, London, United Kingdom - June 24 - 27, 2013*, pages 207–216. ACM, 2013. doi:10.1145/2494603.2480311.

[20] E. Barboni, S. Conversy, D. Navarre, and P. A. Palanque. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In G. J. Doherty and A. Blandford, editors, *Interactive Systems. Design, Specification, and Verification, 13th International Workshop, DSVIS 2006, Dublin, Ireland, July 26-28, 2006. Revised Papers*, volume 4323 of *Lecture Notes in Computer Science*, pages 25–38. Springer, 2006. doi:10.1007/978-3-540-69554-7_3.

[21] P. Barlatier and R. Dapoigny. A type-theoretical approach for ontologies: The case of roles. *Appl. Ontology*, 7(3):311–356, 2012. doi:10.3233/AO-2012-0113.

[22] D. Barrell, E. Dimmer, R. P. Huntley, D. Binns, C. O'Donovan, and R. Apweiler. The GOA database in 2009 - an integrated gene ontology annotation resource. *Nucleic Acids Res.*, 37(Database-Issue):396–403, 2009. doi:10.1093/nar/gkn803.

[23] H. Barringer, J. H. Cheng, and C. B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251–269, 1984. doi:10.1007/BF00264250.

[24] C. Bartolini, A. Giurgiu, G. Lenzini, and L. Robaldo. A Framework to Reason about the Legal Compliance of Security Standards. In *In Proceedings of the Tenth International Workshop on Juris-informatics (JURISIN)*, 2016. URL: https://tinyurl.com/4mutvd72.

[25] N. Belaid, S. Jean, Y. Ait Ameur, and J. Rainaud. An Ontology and Indexation based Management of Services and Workflows Application to Geological Modeling. *Int. J. Electron. Bus. Manag.*, 9(4):296–309, 2011. URL: http://ijebm.ie.nthu.edu.tw/IJEBM_Web/IJEBM_static/Paper-V9_N4/A02.pdf.

[26] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.

[27] D. Bjørner. *Software Engineering 1 - Abstraction and Modelling*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. doi:10.1007/3-540-31288-9.

[28] D. Bjørner. *Software Engineering 2 - Specification of Systems and Languages*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. doi:10.1007/978-3-540-33193-3.

[29] D. Bjørner. *Software Engineering 3 - Domains, Requirements, and Software Design*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. doi:10.1007/3-540-33653-2.

[30] D. Bjørner. Domain Analysis & Description - The Implicit and Explicit Semantics Problem. In R. Laleau, D. Méry, S. Nakajima, and E. Troubitsyna, editors, *Proceedings Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX) and Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD), IMPEX/FM&MDD 2017, Xi'an, China, 16th November 2017*, volume 271 of *EPTCS*, pages 1–23, 2017. doi:10.4204/EPTCS.271.1.

[31] D. Bjørner. Manifest domains: analysis and description. *Formal Aspects Comput.*, 29(2):175–225, 2017. doi:10.1007/s00165-016-0385-z.

[32] D. Bjørner. Domain Analysis and Description Principles, Techniques, and Modelling Languages. *ACM Trans. Softw. Eng. Methodol.*, 28(2):8:1–8:67, 2019. doi:10.1145/3295738.

[33] D. Bjørner and A. Eir. Compositionality: Ontology and Mereology of Domains. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59. Springer, 2010. doi:10.1007/978-3-642-11512-7_3.

[34] D. A. Boehm-Davis, R. E. Curry, E. L. Wiener, and R. L. Harrison. Human factors of flight-deck automation: Report on a NASA-industry workshop. *Ergonomics*, 26(10):953–961, 1983. doi:10.1080/00140138308963424.

[35] M. L. Bolton, R. I. Siminiceanu, and E. J. Bass. A Systematic Approach to Model Checking Human-Automation Interaction Using Task Analytic Models. *IEEE Trans. Syst. Man Cybern. Part A*, 41(5):961–976, 2011. doi:10.1109/TSMCA.2011.2109709.

[36] K. Bontcheva, V. Tablan, D. Maynard, and H. Cunningham. Evolving GATE to meet new challenges in language engineering. *Nat. Lang. Eng.*, 10(3-4):349–373, 2004. doi:10.1017/S1351324904003468.

[37] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C recommendation, W3C, February 2004. URL: http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[38] J. Broekstra and A. Kampman. An RDF Query and Transformation Language. In S. Staab and H. Stuckenschmidt, editors, *Semantic Web and Peer-to-Peer - Decentralized Management and Exchange of Knowledge and Information*, pages 23–39. Springer, 2006. doi:10.1007/3-540-28347-1_2.

[39] M. J. Butler and I. Maamria. Practical Theory Extension in Event-B. In Z. Liu, J. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013. doi:10.1007/978-3-642-39698-4_5.

[40] D. Calegari, T. Mossakowski, and N. Szasz. Heterogeneous verification in the context of model driven engineering. *Sci. Comput. Program.*, 126:3–30, 2016. doi:10.1016/j.scico.2016.02.003.

[41] J. C. Campos, C. Fayollas, M. D. Harrison, C. Martinie, P. Masci, and P. A. Palanque. Supporting the Analysis of Safety Critical User Interfaces: An Exploration of Three Formal Tools. *ACM Trans. Comput. Hum. Interact.*, 27(5):35:1–35:48, 2020. doi:10.1145/3404199.

[42] J. C. Campos and M. D. Harrison. Formal Verification of Interactive Computing Systems: Opportunities, Challenges. In B. Weyers and J. Bowen, editors, *Joint Proceedings HCI Engineering 2019 - Methods and Tools for Advanced Interactive Systems and Integration of Multiple Stakeholder Viewpoints co-located with 11th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2019), Valencia, Spain, June 18, 2019*, volume 2503 of *CEUR Workshop Proceedings*, pages 69–75. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2503/paper1_11.pdf.

[43] J. Carmona, B. F. van Dongen, and M. Weidlich. Conformance Checking: Foundations, Milestones and Challenges. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 155–190. Springer, 2022. doi:10.1007/978-3-031-08848-3_5.

[44] A. Chebieb and Y. Ait Ameur. A formal model for plastic human computer interfaces. *Frontiers Comput. Sci.*, 12(2):351–375, 2018. doi:10.1007/s11704-016-5460-3.

[45] A. Chebotko, Y. Deng, S. Lu, F. Fotouhi, and A. Aristar. An ontology-based multimedia annotator for the semantic web of language engineering. *Int. J. Semantic Web Inf. Syst.*, 1(1):50–67, 2005. doi:10.4018/jswis.2005010104.

[46] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. doi:10.1007/3-540-45657-0_29.

[47] M. Codescu, E. Kuksa, O. Kutz, T. Mossakowski, and F. Neuhaus. Ontohub: A semantic repository for heterogeneous ontologies. *CoRR*, abs/1612.05028, 2016. URL: http://arxiv.org/abs/1612.05028, arXiv:1612.05028.

[48] D. Connolly, I. Horrocks, D. McGuinness, F. Patel-Schneider, and A. Stein. DAML+OIL Reference Description. *World Wide Web Consortium*, 2001. URL: https://www.w3.org/TR/2001/NOTE-daml+oil-reference-20011218.

[49] R. Dapoigny and P. Barlatier. Modeling Ontological Structures with Type Classes in Coq. In H. D. Pfeiffer, D. I. Ignatov, J. Poelmans, and N. Gadiraju, editors, *Conceptual Structures for STEM Research and Education, 20th International Conference on Conceptual Structures, ICCS 2013, Mumbai, India, January 10-12, 2013. Proceedings*, volume 7735 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2013. doi:10.1007/978-3-642-35786-2_11.

[50] R. Dapoigny and P. Barlatier. Formalizing context for domain ontologies in coq. In P. Brézillon and A. J. Gonzalez, editors, *Context in Computing - A Cross-Disciplinary Approach for Modeling the Real World*, pages 437–454. Springer, 2014. doi:10.1007/978-1-4939-1887-4_27.

[51] B. d'Ausbourg. Using Model Checking for the Automatic Validation of User Interface Systems. In P. Markopoulos and P. Johnson, editors, *Design, Specification and Verification of Interactive Systems'98, Proceedings*

of the Fifth International Eurographics Workshop, June 3-5, 1998, Abingdon, United Kingdom, Volume 1, Eurographics, pages 242–260. Springer, 1998. doi:10.1007/978-3-7091-3693-5_16.

[52] B. d'Ausbourg, G. Durrieu, and P. Roché. Deriving a Formal Model of an Interactive System from its UIL Descriptionin order to Verify and Test its Behaviour. In F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems'96, Proceedings of the Third International Eurographics Workshop, June 5-7, 1996, Namur, Belgium*, Eurographics, pages 105–122. Springer, 1996. doi:10.1007/978-3-7091-7491-3_6.

[53] B. d'Ausbourg, C. Seguin, G. Durrieu, and P. Roché. Helping the Automated Validation Process of User Interfaces Systems. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998*, pages 219–228. IEEE Computer Society, 1998. doi:10.1109/ICSE.1998.671121.

[54] H. Dehainsala, G. Pierra, and L. Bellatreche. OntoDB: An Ontology-Based Database for Data Intensive Applications. In K. Ramamohanarao, P. R. Krishna, M. K. Mohania, and E. Nantajeewarawat, editors, *Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings*, volume 4443 of *Lecture Notes in Computer Science*, pages 497–508. Springer, 2007. doi:10.1007/978-3-540-71703-4_43.

[55] F. Dehais, C. Tessier, and L. Chaudron. GHOST: Experimenting Conflicts Countermeasures in the Pilot's Activity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, page 163–168, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc. URL: https://dl.acm.org/doi/10.5555/1630659.1630682.

[56] S. Després and S. Szulman. Terminae Method and Integration Process for Legal Ontology Building. In M. Ali and R. Dapoigny, editors, *Advances in Applied Artificial Intelligence, 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2006, Annecy, France, June 27-30, 2006, Proceedings*, volume 4031 of *Lecture Notes in Computer Science*, pages 1014–1023. Springer, 2006. doi:10.1007/11779568_108.

[57] D. Distante, M. Winckler, R. Bernhaupt, J. Bowen, J. C. Campos, F. Müller, P. A. Palanque, J. V. den Bergh, B. Weyers, and A. Voit. Trends on engineering interactive systems: an overview of works presented in workshops at EICS 2019. In J. I. Panach, J. Vanderdonckt, and O. Pastor, editors, *Proceedings of the ACM SIGCHI Symposium on Engineering*

Interactive Computing Systems, EICS 2019, Valencia, Spain, June 18-21, 2019, pages 22:1–22:6. ACM, 2019. doi:10.1145/3319499.3335655.

[58] G. Dupont, Y. Ait Ameur, M. Pantel, and N. K. Singh. Handling Refinement of Continuous Behaviors: A Proof Based Approach with Event-B. In D. Méry and S. Qin, editors, *2019 International Symposium on Theoretical Aspects of Software Engineering, TASE 2019, Guilin, China, July 29-31, 2019*, pages 9–16. IEEE, 2019. doi:10.1109/TASE.2019.00-25.

[59] G. Dupont, Y. Ait Ameur, M. Pantel, and N. K. Singh. Formally Verified Architecture Patterns of Hybrid Systems Using Proof and Refinement with Event-B. In A. Raschke, D. Méry, and F. Houdek, editors, *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings*, volume 12071 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 2020. doi:10.1007/978-3-030-48077-6_12.

[60] G. Dupont, Y. Ait Ameur, N. K. Singh, and M. Pantel. Event-B Hybridation: A Proof and Refinement-based Framework for Modelling Hybrid Systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):35:1–35:37, 2021. doi:10.1145/3448270.

[61] W. Emmerich, A. Finkelstein, C. Montangero, S. Antonelli, S. Armitage, and R. Stevens. Managing Standards Compliance. *IEEE Trans. Software Eng.*, 25(6):826–851, 1999. doi:10.1109/32.824413.

[62] ESA. ECSS-Q-HB-30-03A – Human dependability handbook, 2015. URL: https://tinyurl.com/y2k4jy7m.

[63] EUROCONTROL. Airborne Collision Avoidance System (ACAS) guide, 12 2017. URL: https://www.eurocontrol.int/publication/airborne-collision-avoidance-system-acas-guide.

[64] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. URL: https://tinyurl.com/8nutw63c.

[65] C. Fayollas, C. Martinie, P. A. Palanque, E. Barboni, R. Fahssi, and A. Hamon. Exploiting Action Theory as a Framework for Analysis and Design of Formal Methods Approaches: Application to the CIRCUS Integrated Development Environment. In B. Weyers, J. Bowen, A. J. Dix, and P. A. Palanque, editors, *The Handbook of Formal Methods in Human-Computer Interaction*, pages 465–504. Springer International Publishing, 2017. doi:10.1007/978-3-319-51838-1_17.

[66] R. T. C. for Aeronautics. Software Considerations in Airborne Systems and Equipment Certification, 2012. URL: https://tinyurl.com/4h7k6vnz.

[67] E. Gabrilovich and S. Markovitch. Wikipedia-based Semantic Interpretation for Natural Language Processing. *J. Artif. Intell. Res.*, 34:443–498, 2009. doi:10.1613/jair.2669.

[68] N. Ge, A. Dieumegard, E. Jenn, B. d'Ausbourg, and Y. Ait Ameur. Formal development process of safety-critical embedded human machine interface systems. In F. Mallet, M. Zhang, and E. Madelaine, editors, *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*, pages 1–8. IEEE Computer Society, 2017. doi:10.1109/TASE.2017.8285636.

[69] R. Geniet and N. K. Singh. Refinement Based Formal Development of Human-Machine Interface. In M. Mazzara, I. Ober, and G. Salaün, editors, *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, volume 11176 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2018. doi:10.1007/978-3-030-04771-9_19.

[70] J. Goodenough, C. Weinstock, and A. Klein. Toward a Theory of Assurance Case Confidence. Technical Report CMU/SEI-2012-TR-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2012. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28067.

[71] W. D. Gray, P. A. Palanque, and F. Paternò. Introduction to the special issue on interface issues and designs for safety-critical interactive systems: when there is no room for user error. *ACM Trans. Comput. Hum. Interact.*, 6(4):309–310, 1999. doi:10.1145/331490.332826.

[72] S. Grigorova and T. S. E. Maibaum. Argument Evaluation in the Context of Assurance Case Confidence Modeling. In *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, pages 485–490. IEEE Computer Society, 2014. doi:10.1109/ISSREW.2014.87.

[73] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. URL: https://doi.org/10.1006/knac.1993.1008.

[74] J. Guiochet, Q. A. D. Hoang, and M. Kaâniche. A Model for Safety Case Confidence Assessment. In F. Koornneef and C. van Gulijk, editors, *Computer Safety, Reliability, and Security - 34th International Conference, SAFECOMP 2015 Delft, The Netherlands, September 23-25, 2015. Proceedings*, volume 9337 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2015. doi:10.1007/978-3-319-24255-2_23.

[75] V. Haarslev and R. Möller. Description of the RACER System and its Applications. In C. A. Goble, D. L. McGuinness, R. Möller, and P. F.

Patel-Schneider, editors, *Working Notes of the 2001 International Description Logics Workshop (DL-2001), Stanford, CA, USA, August 1-3, 2001*, volume 49 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001. URL: https://ceur-ws.org/Vol-49/HaarslevMoeller-132start.ps.

[76] K. Hacid and Y. Ait Ameur. Strengthening MDE and Formal Design Models by References to Domain Ontologies. A Model Annotation Based Approach. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *Lecture Notes in Computer Science*, pages 340–357, 2016. doi:10.1007/978-3-319-47166-2_24.

[77] K. Hacid and Y. Ait Ameur. Handling Domain Knowledge in Design and Analysis of Engineering Models. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 74, 2017. doi:10.14279/tuj.eceasst.74.1045.

[78] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, 1991. doi:10.1109/5.97300.

[79] S. Handschuh and S. Staab. CREAM: CREAting Metadata for the Semantic Web. *Comput. Networks*, 42(5):579–598, 2003. doi:10.1016/S1389-1286(03)00226-3.

[80] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In R. Volz, S. Decker, and I. F. Cruz, editors, *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003. URL: https://ceur-ws.org/Vol-89/harris-et-al.pdf.

[81] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction*. Cambridge University Press, USA, 1990. URL: https://dl.acm.org/doi/10.5555/94033.

[82] M. D. Harrison. Examples of the Application of Formal Methods to Interactive Systems. In E. Sekerinski, N. Moreira, J. N. Oliveira, D. Ratiu, R. Guidotti, M. Farrell, M. Luckcuck, D. Marmsoler, J. C. Campos, T. Astarte, L. Gonnord, A. Cerone, L. Couto, B. Dongol, M. Kutrib, P. Monteiro, and D. Delmas, editors, *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, volume 12232 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 2019. doi:10.1007/978-3-030-54994-7_31.

[83] M. D. Harrison, P. Masci, and J. C. Campos. Verification Templates for the Analysis of User Interface Software Design. *IEEE Trans. Software Eng.*, 45(8):802–822, 2019. doi:10.1109/TSE.2018.2804939.

[84] I. Harrow, R. Balakrishnan, E. Jimenez-Ruiz, S. Jupp, J. Lomax, J. Reed, M. Romacker, C. Senger, A. Splendiani, J. Wilson, and P. Woollard. Ontology mapping for semantically enabled applications. *Drug Discovery Today*, 24(10):2068–2075, 2019. URL: https://www.sciencedirect.com/science/article/pii/S1359644618304215, doi:10.1016/j.drudis.2019.05.020.

[85] B. Henderson-Sellers. *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Springer Briefs in Computer Science. Springer, 2012. doi:10.1007/978-3-642-29825-7.

[86] T. S. Hoang and J. Abrial. Reasoning about Liveness Properties in Event-B. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, volume 6991 of *Lecture Notes in Computer Science*, pages 456–471. Springer, 2011. doi:10.1007/978-3-642-24559-6_31.

[87] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artif. Intell.*, 194:28–61, 2013. doi:10.1016/j.artint.2012.06.001.

[88] IEC-61360-4. Standard data element types with associated classification scheme for electric components - part 4 : Iec reference collection of standard data element types, component classes and terms (withdrawn). Technical report, 1999.

[89] IEC 62304. IEC 62304:2006 - Medical Device Software - Software Life Cycle Processes, 5 2006.

[90] M. Jackson and P. Zave. Domain descriptions. In *Proceedings of IEEE International Symposium on Requirements Engineering, RE 1993, San Diego, California, USA, January 4-6, 1993*, pages 56–64. IEEE Computer Society, 1993. doi:10.1109/ISRE.1993.324836.

[91] M. Jackson and P. Zave. Deriving Specifications from Requirements: An Example. In D. E. Perry, R. Jeffery, and D. Notkin, editors, *17th International Conference on Software Engineering, Seattle, Washington, USA, April 23-30, 1995, Proceedings*, pages 15–24. ACM, 1995. doi:10.1145/225014.225016.

[92] R. Jacquart, editor. *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, volume 156 of *IFIP*. Kluwer/Springer, 2004.

[93] S. Jean, Y. Ait Ameur, and G. Pierra. Querying Ontology Based Database Using OntoQL (An Ontology Query Language). In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International*

Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I, volume 4275 of *Lecture Notes in Computer Science*, pages 704–721. Springer, 2006. doi:10.1007/11914853_43.

[94] S. Jean, G. Pierra, and Y. Ait Ameur. Domain Ontologies: A Database-Oriented Analysis. In J. Filipe, J. Cordeiro, and V. Pedrosa, editors, *Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers*, volume 1 of *Lecture Notes in Business Information Processing*, pages 238–254. Springer, 2006. doi:10.1007/978-3-540-74063-6_19.

[95] C. B. Jones. Partial Functions and Logics: A Warning. *Inf. Process. Lett.*, 54(2):65–67, 1995. doi:10.1016/0020-0190(95)00042-B.

[96] H. Kaiya and M. Saeki. Using Domain Ontology as Domain Knowledge for Requirements Elicitation. In *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*, pages 186–195. IEEE Computer Society, 2006. doi:10.1109/RE.2006.72.

[97] H. Kaiya, Y. Shimizu, H. Yasui, K. Kaijiri, and M. Saeki. Enhancing Domain Knowledge for Requirements Elicitation with Web Mining. In J. Han and T. D. Thu, editors, *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, pages 3–12. IEEE Computer Society, 2010. doi:10.1109/APSEC.2010.11.

[98] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In D. Lassner, D. D. Roure, and A. Iyengar, editors, *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, May 7-11, 2002, Honolulu, Hawaii, USA*, pages 592–603. ACM, 2002. doi:10.1145/511446.511524.

[99] T. Kelly. *Arguing Safety – A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, 09 1998. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.285977.

[100] S. Kherroubi and D. Méry. Contextualization and Dependency in State-Based Modelling - Application to Event-B. In Y. Ouhammou, M. Ivanovic, A. Abelló, and L. Bellatreche, editors, *Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings*, volume 10563 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2017. doi:10.1007/978-3-319-66854-3_11.

[101] S. Kherroubi and D. Méry. Contextualization and Dependency in State-Based Modelling - Application to Event-B. In Y. Ouhammou, M. Ivanovic, A. Abelló, and L. Bellatreche, editors, *Model and Data Engineering - 7th*

International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings, volume 10563 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2017. doi:10.1007/978-3-319-66854-3_11.

[102] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, volume 3298 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2004. doi:10.1007/978-3-540-30475-3_17.

[103] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: semantic annotations for WSDL and XML schema. *IEEE Internet Comput.*, 11(6):60–67, 2007. doi:10.1109/MIC.2007.134.

[104] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, aug 1988. URL: https://dl.acm.org/doi/10.5555/50757.50759.

[105] C. Larman and V. R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6):47–56, 2003. doi:10.1109/MC.2003.1204375.

[106] V. Lecrubier. *Un langage formel pour la conception, la spécification et la vérification d'interfaces homme-machine embarquées critiques. (A formal language for designing, specifying and verifying critical embedded human machine interfaces)*. PhD thesis, Institut supérieur de l'aéronautique et de l'espace, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01455466.

[107] M. Leuschel. Fast and Effective Well-Definedness Checking. In B. Dongol and E. Troubitsyna, editors, *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*, volume 12546 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2020. doi:10.1007/978-3-030-63461-2_4.

[108] H. J. Levesque. A Logic of Implicit and Explicit Belief. In R. J. Brachman, editor, *Proceedings of the National Conference on Artificial Intelligence. Austin, TX, USA, August 6-10, 1984*, pages 198–202. AAAI Press, 1984. URL: http://www.aaai.org/Library/AAAI/1984/aaai84-038.php.

[109] Y. Lu, H. Panetto, Y. Ni, and X. Gu. Ontology alignment for networked enterprise information system interoperability in supply chain environment. *Int. J. Comput. Integr. Manuf.*, 26(1-2):140–151, 2013. doi:10.1080/0951192X.2012.681917.

[110] H. Luong, T. Lambolais, and A. Courbis. Implementation of the Conformance Relation for Incremental Development of Behavioural Models. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2008. doi:10.1007/978-3-540-87875-9_26.

[111] A. Mashkoor and J. Jacquot. Utilizing Event-B for domain engineering: a critical analysis. *Requir. Eng.*, 16(3):191–207, 2011. doi:10.1007/s00766-011-0120-5.

[112] L. S. Mastella, Y. Ait Ameur, S. Jean, M. Perrin, and J. Rainaud. Semantic Exploitation of Engineering Models: An Application to Oilfield Models. In A. P. Sexton, editor, *Dataspace: The Final Frontier, 26th British National Conference on Databases, BNCOD 26, Birmingham, UK, July 7-9, 2009. Proceedings*, volume 5588 of *Lecture Notes in Computer Science*, pages 203–207. Springer, 2009. doi:10.1007/978-3-642-02843-4_22.

[113] J. McDermid and K. Ripken. Life Cycle Support in the Ada Environment. *Ada Lett.*, III(1):57–62, jul 1983. doi:10.1145/998373.998379.

[114] I. Mendil. A framework for critical interactive system formal modelling and analysis. In A. Raschke, D. Méry, and F. Houdek, editors, *Rigorous State-Based Methods*, pages 423–426, Cham, 2020. Springer International Publishing.

[115] I. Mendil, Y. Ait Ameur, N. K. Singh, G. Dupont, D. Méry, and P. A. Palanque. Formal domain-driven system development in Event-B: Application to interactive critical systems. *Journal of Systems Architecture: Embedded Software Design (JSA)*, 135:102798, 2023. doi:10.1016/j.sysarc.2022.102798.

[116] I. Mendil, Y. Ait Ameur, N. K. Singh, D. Méry, and P. A. Palanque. Leveraging Event-B Theories for Handling Domain Knowledge in Design Models. In S. Qin, J. Woodcock, and W. Zhang, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 7th International Symposium, SETTA 2021, Beijing, China, November 25-27, 2021, Proceedings*, volume 13071 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2021. doi:10.1007/978-3-030-91265-9_3.

[117] I. Mendil, Y. Ait Ameur, N. K. Singh, D. Méry, and P. A. Palanque. Standard Conformance-by-Construction with Event-B. In A. Lluch-Lafuente and A. Mavridou, editors, *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*, volume 12863 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2021. doi:10.1007/978-3-030-85248-1_8.

[118] I. Mendil, P. Rivière, Y. Ait Ameur, N. K. Singh, D. Méry, and P. A. Palanque. Non-Intrusive Annotation-Based Domain-Specific Analysis to Certify Event-B Models Behaviours. In *29th Asia-Pacific Software Engineering Conference, APSEC 2022, Virtual Event, Japan, December 6-9, 2022*, pages 129–138. IEEE, 2022. doi:10.1109/APSEC57359.2022.00025.

[119] I. Mendil, N. K. Singh, Y. Ait Ameur, D. Méry, and P. A. Palanque. An Integrated Framework for the Formal Analysis of Critical Interactive Systems. In *27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020*, pages 139–148. IEEE, 2020. doi:10.1109/APSEC51365.2020.00022.

[120] D. Méry, R. Sawant, and A. Tarasyuk. Integrating domain-based features into event-b: A nose gear velocity case study. In L. Bellatreche and Y. Manolopoulos, editors, *Model and Data Engineering - 5th International Conference, MEDI 2015, Rhodes, Greece, September 26-28, 2015, Proceedings*, volume 9344 of *Lecture Notes in Computer Science*, pages 89–102. Springer, 2015. doi:10.1007/978-3-319-23781-7_8.

[121] D. Méry and N. K. Singh. Automatic code generation from Event-B models. In H. Q. Thang and D. K. Tran, editors, *Proceedings of the 2011 Symposium on Information and Communication Technology, SoICT 2011, Hanoi, Viet Nam, October 13-14, 2011*, pages 179–188. ACM, 2011. doi:10.1145/2069216.2069252.

[122] S. Millett and N. Tune. *Patterns, principles, and practices of domain-driven design*. John Wiley & Sons, 2015. URL: https://tinyurl.com/5znd7edw.

[123] L. Mohand-Oussaïd and I. Ait Sadoune. Formal Modelling of Domain Constraints in Event-B. In Y. Ouhammou, M. Ivanovic, A. Abelló, and L. Bellatreche, editors, *Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings*, volume 10563 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2017. doi:10.1007/978-3-319-66854-3_12.

[124] L. Mohand Oussaid and I. Ait Sadoune. OntoEventB : Un outil pour la modélisation des ontologies dans B Événementiel. In *AFADL 2017*, pages 117–121, Montpellier, France, June 2017. URL: https://hal.archives-ouvertes.fr/hal-01546065.

[125] T. Mossakowski. The Distributed Ontology, Model and Specification Language - DOL. In P. James and M. Roggenbach, editors, *Recent Trends in Algebraic Development Techniques - 23rd IFIP WG 1.3 International Workshop, WADT 2016, Gregynog, UK, September 21-24, 2016, Revised Selected Papers*, volume 10644 of *Lecture Notes in Computer Science*, pages 5–10. Springer, 2016. doi:10.1007/978-3-319-72044-9_2.

[126] B. Motik. KAON2 - scalable reasoning over ontologies with large data sets. *ERCIM News*, 2008(72), 2008. URL: http://ercim-news.ercim.eu/kaon2-scalable-reasoning-over-ontologies-with-large-data-sets.

[127] K. Munir and M. Sheraz Anjum. The use of ontologies for effective knowledge modelling and information retrieval. *Applied Computing and Informatics*, 14(2):116–126, 2018. doi:10.1016/j.aci.2017.07.003.

[128] S. Nair, J. L. de la Vara, M. Sabetzadeh, and D. Falessi. Evidence management for compliance of critical systems with safety standards: A survey on the state of practice. *Inf. Softw. Technol.*, 60:1–15, 2015. doi:10.1016/j.infsof.2014.12.002.

[129] D. Navarre and P. A. Palanque. The future of design specification and verification of safety critical interactive systems.: can our systems be sure (safe, usable, reliable and evolvable)? In T. C. N. Graham, G. Calvary, and P. D. Gray, editors, *Proceedings of the 1st ACM SIGCHI symposium on Engineering Interactive Computing System , EICS 2009, Pittsburgh, PA, USA, July 15-17, 2009*, pages 155–156. ACM, 2009. doi:10.1145/1570433.1570463.

[130] D. Navarre, P. A. Palanque, R. Bastide, and O. Sy. A Model-Based Tool for Interactive Prototyping of Highly Interactive Applications. In *12th IEEE International Workshop on Rapid System Prototyping (RSP 2001), 25-27 June 2001, Monterey, CA, USA*, pages 136–141. IEEE Computer Society, 2001. doi:10.1109/IWRSP.2001.933851.

[131] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. URL: https://doi.org/10.1007/3-540-45949-9.

[132] D. A. Norman and S. W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., USA, 1986. URL: https://dl.acm.org/doi/10.5555/576915.

[133] N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical report, march 2001. URL: http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html.

[134] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992. doi:10.1007/3-540-55602-8_217.

[135] P. Palanque and F. Paterno. *Formal Methods in Human-Computer Interaction*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997. URL: https://dl.acm.org/doi/book/10.5555/550442.

[136] D. L. Parnas and J. Madey. Functional Documents for Computer Systems. *Sci. Comput. Program.*, 25(1):41–61, 1995. doi:10.1016/0167-6423(95)96871-J.

[137] G. Pierra. The PLIB ontology-based approach to data integration. In R. Jacquart, editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, volume 156 of *IFIP*, pages 13–18. Kluwer/Springer, 2004. doi:10.1007/978-1-4020-8157-6_2.

[138] G. Pierra. Context Representation in Domain Ontologies and Its Use for Semantic Integration of Data. *J. Data Semant.*, 10:174–211, 2008. doi:10.1007/978-3-540-77688-8_6.

[139] G. Pierra and E. Sardet. *ISO 13584-32:2010 – Industrial automation systems and integration – Parts library – Part 32: Implementation resources: OntoML: Product ontology markup language*. ISO, 2010. URL: https://hal.archives-ouvertes.fr/hal-03368894.

[140] E. Prud'hommeaux. SPARQL query language for RDF, W3C recommendation. 2008. URL: http://www.w3.org/TR/rdf-sparql-query/.

[141] S. Ranville and F. Bachmann. How to Meet Compliance to Software Architecture Design Principles. *SAE Technical Paper Series*, 2019. doi:10.4271/2019-01-1040.

[142] M. Rauterberg. An Iterative-Cyclic Software Process Model. In *SEKE'92, The 4th International Conference on Software Engineering and Knowledge Engineering, June, 15-20 1992, Capri, Italy*, pages 600–607. IEEE Computer Society, 1992. doi:10.1109/SEKE.1992.227899.

[143] P. Rivière, N. K. Singh, and Y. Ait Ameur. EB4EB: A Framework for Reflexive Event-B. In *26th International Conference on Engineering of Complex Computer Systems, ICECCS 2022, Hiroshima, Japan, March 26-30, 2022*, pages 71–80. IEEE, 2022. doi:10.1109/ICECCS54210.2022.00017.

[144] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, page 328–338, Washington, DC, USA, 1987. IEEE Computer Society Press. URL: https://dl.acm.org/doi/10.5555/41765.41801.

[145] J. Rushby. The Interpretation and Evaluation of Assurance Cases. Technical Report SRI-CSL-15-01, Computer Science Laboratory, SRI International, Menlo Park, CA, July 2015. Available at http://www.csl.sri.com/users/rushby/papers/sri-csl-15-1-assurance-cases.pdf.

[146] D. Sannella and A. Tarlecki. *Structured specifications*, pages 229–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-17336-3_5.

[147] B. Shneiderman, C. Plaisant, M. S. Cohen, S. Jacobs, and N. Elmqvist. *Designing the User Interface - Strategies for Effective Human-Computer Interaction, 6th Edition.* Pearson, 2016. URL: http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-013438038X,00.html.

[148] N. K. Singh. *EB2ALL: An Automatic Code Generation Tool*, pages 105–141. Springer London, London, 2013. doi:10.1007/978-1-4471-5260-6_7.

[149] N. K. Singh. *Using Event-B for Critical Device Software Systems.* Springer, 2013. doi:10.1007/978-1-4471-5260-6.

[150] N. K. Singh, Y. Ait Ameur, R. Geniet, D. Méry, and P. A. Palanque. On the Benefits of Using MVC Pattern for Structuring Event-B Models of WIMP Interactive Applications. *Interact. Comput.*, 33(1):92–114, 2021. doi:10.1093/iwcomp/iwab016.

[151] N. K. Singh, Y. Ait Ameur, I. Mendil, D. Méry, D. Navarre, P. Palanque, and M. Pantel. F3FLUID: A formal framework for developing safety-critical interactive systems in FLUID. *Journal of Software: Evolution and Process*, page e2439, 2022. doi:10.1002/smr.2439.

[152] N. K. Singh, Y. Ait Ameur, and D. Méry. Formal Ontological Analysis for Medical Protocols. In Y. Ait Ameur, S. Nakajima, and D. Méry, editors, *Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems: Communications of NII Shonan Meetings*, pages 83–107, Singapore, 2021. Springer Singapore. doi:10.1007/978-981-15-5054-6_5.

[153] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *J. Web Semant.*, 5(2):51–53, 2007. doi:10.1016/j.websem.2007.03.004.

[154] B. Stoddart, S. Dunne, and A. Galloway. Undefined Expressions and Logic in Z and B. *Formal Methods Syst. Des.*, 15(3):201–215, 1999. doi:10.1023/A:1008797018928.

[155] ED 143 - Minimum Operational Performance Standards for Traffic Alert and Collision Avoidance System II (TCAS II), 2013. URL: https://standards.globalspec.com/std/1609213/EUROCAEED143.

[156] S. E. Toulmin. *The Uses of Argument.* Cambridge University Press, 2 edition, 2003. doi:10.1017/CBO9780511840005.

[157] J. Trinkunas. A graph oriented model for ontology transformation into conceptual data model. *Information technology and control*, 36, 01 2007. URL: https://etalpykla.vilniustech.lt/handle/123456789/63386.

[158] V. S. Uren, P. Cimiano, J. Iria, S. Handschuh, M. Vargas-Vera, E. Motta, and F. Ciravegna. Semantic annotation for knowledge management: Requirements and a survey of the state of the art. *J. Web Semant.*, 4(1):14–28, 2006. doi:10.1016/j.websem.2005.10.002.

[159] M. Uschold. Where Are the Semantics in the Semantic Web? *AI Mag.*, 24(3):25–36, 2003. doi:10.1609/aimag.v24i3.1716.

[160] A. van Dam. Post-WIMP User Interfaces. *Commun. ACM*, 40(2):63–67, 1997. doi:10.1145/253671.253708.

[161] A. van Lamsweerde and L. Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Trans. Software Eng.*, 24(12):1089–1114, 1998. doi:10.1109/32.738341.

[162] I. Vistbakka and E. Troubitsyna. Deriving Implicit Security Requirements in Safety-Explicit Formal Development of Control Systems. In Y. Ait Ameur, S. Nakajima, and D. Méry, editors, *Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems: Communications of NII Shonan Meetings*, pages 109–130, Singapore, 2021. Springer Singapore. doi:10.1007/978-981-15-5054-6_6.

[163] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. URL: http://www.w3.org/TR/owl2-overview/.

[164] A. Wassyng, P. Joannou, M. Lawford, T. Maibaum, and N. K. Singh. New standards for trustworthy cyber-physical systems. *Trustworthy Cyber-Physical Systems Engineering*, pages 337–368, 2016. URL: https://tinyurl.com/39xfkx8x.

[165] A. Wassyng, N. K. Singh, M. Geven, N. Proscia, H. Wang, M. Lawford, and T. Maibaum. Can Product-Specific Assurance Case Templates Be Used as Medical Device Standards? *IEEE Des. Test*, 32(5):45–55, 2015. doi:10.1109/MDAT.2015.2462720.

[166] C. D. Wickens, J. Lee, Y. D. Liu, and S. Gordon-Becker. *Introduction to Human Factors Engineering (2nd Edition)*. Prentice-Hall, Inc., USA, 2013. URL: https://tinyurl.com/4yuw9xc6.

[167] L. Wiener. *Human Factors of Advanced Technology (glass Cockpit) Transport Aircraft*. NASA contractor report. National Aeronautics and Space Administration, 1989. URL: https://ntrs.nasa.gov/citations/19890016609.

[168] K. Wilkinson. Jena property table implementation. In *Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'06)*, pages 35–46, 2006.

[169] E. Williams. Airborne Collision Avoidance System. In T. Cant, editor, *Ninth Australian Workshop on Safety-Related Programmable Systems (SCS 2004)*, volume 47 of *CRPIT*, pages 97–110, Brisbane, Australia, 2004. ACS. URL: https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV47Williams.pdf.

[170] P. Zave. Classification of Research Efforts in Requirements Engineering. *ACM Comput. Surv.*, 29(4):315–321, 1997. doi:10.1145/267580.267581.

[171] P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997. doi:10.1145/237432.237434.

[172] D. S. Zayas, A. Monceaux, and Y. Ait Ameur. Knowledge Models to Reduce the Gap between Heterogeneous Models: Application to Aircraft Systems Engineering. In R. Calinescu, R. F. Paige, and M. Z. Kwiatkowska, editors, *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March 2010*, pages 355–360. IEEE Computer Society, 2010. doi:10.1109/ICECCS.2010.35.

[173] Y. Zhao, Z. Yang, D. Sanán, and Y. Liu. Event-based formalization of safety-critical operating system standards: An experience report on ARINC 653 using Event-B. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 281–292. IEEE Computer Society, 2015. doi:10.1109/ISSRE.2015.7381821.

Appendices

Appendix A

Meta-Modelling Theories

This appendix contains:

A.1 `OntologiesTheory` - Ontology Modelling Language 162
A.2 `EvtBTheo` - Event-B Meta-Theory 163

A.1 `OntologiesTheory` - Ontology Modelling Language

Intentionally removed the Event-B code

A.2 `EvtBTheo` - Event-B Meta-Theory

Intentionally removed the Event-B code

Appendix B

Domain Theories

This appendix contains:

B.1 DisplayabilityTheory - Displayability Domain Theory 166
B.2 ARINC661Theory - ARINC 661 Standard Domain theory 167
B.3 Domain-Specific Behaviour Analysis 168
B.3.1 BehaviouralPropertiesTheory - Analysis Operator 168
B.3.2 Theo4Reachability - Analysis Low-Level Terms 168
B.3.3 EvtBManip - Auxiliary Operators 168

B.1 `DisplayabilityTheory` - Displayability Domain Theory

Intentionally removed the Event-B code

B.2 `ARINC661Theory` - ARINC 661 Standard Domain theory

Intentionally removed the Event-B code

B.3 Domain-Specific Behaviour Analysis

B.3.1 `BehaviouralPropertiesTheory` - Analysis Operator

Intentionally removed the Event-B code

B.3.2 `Theo4Reachability` - Analysis Low-Level Terms

Intentionally removed the Event-B code

B.3.3 `EvtBManip` - Auxiliary Operators

Intentionally removed the Event-B code

Appendix C

Case Studies

This appendix contains:

C.1 Temperature Aggregator Case Study Modelling 170
C.1.1 `C_TemperatureContext` - Event-B Context for Units 170
C.1.2 `C_TemperatureMachine` - Machine without Operators 170
C.1.3 `ThermalUnits` - Event-B Theory for Units 170
C.1.4 `T_TemperatureMachine` - Event-B Machine with Operators . . 170
C.2 TCAS Case Study Modelling 171
C.2.1 `InstantiationContext` - Event-B Context for Instantiation . 171
C.2.2 `SetTheoriticOperationsBasedModel` 171
C.2.3 `TheoryOperatorsBasedModel` 171
C.3 ATM Case Study Modelling 172
C.3.1 `ATMEnvironment` - Event-B Context for Constants 172
C.3.2 `ATMUserInterface` - Event-B Machine for ATM Model 172
C.3.3 `ATMmEBModel` - ATM Model Exported to EB4EB 172
C.3.4 `AnnotatedModel` - Annotation and Analysis 172
C.4 WXR Case Study Modelling 173
C.4.1 `WXRModel` - Event-B Machine for WXR Model 173

C.1 Temperature Aggregator Case Study Modelling

C.1.1 `C_TemperatureContext` - Event-B Context for Units

Intentionally removed the Event-B code

C.1.2 `C_TemperatureMachine` - Machine without Operators

Intentionally removed the Event-B code

C.1.3 `ThermalUnits` - Event-B Theory for Units

Intentionally removed the Event-B code

C.1.4 `T_TemperatureMachine` - Event-B Machine with Operators

Intentionally removed the Event-B code

C.2 TCAS Case Study Modelling

C.2.1 `InstantiationContext` - Event-B Context for Instantiation

Intentionally removed the Event-B code

C.2.2 `SetTheoriticOperationsBasedModel`

Intentionally removed the Event-B code

C.2.3 `TheoryOperatorsBasedModel`

Intentionally removed the Event-B code

C.3 ATM Case Study Modelling

C.3.1 `ATMEnvironment` - Event-B Context for Constants

Intentionally removed the Event-B code

C.3.2 `ATMUserInterface` - Event-B Machine for ATM Model

Intentionally removed the Event-B code

C.3.3 `ATMmEBModel` - ATM Model Exported to EB4EB

Intentionally removed the Event-B code

C.3.4 `AnnotatedModel` - Annotation and Analysis

Intentionally removed the Event-B code

C.4 WXR Case Study Modelling

Intentionally removed the Event-B code

C.4.1 `WXRModel` - Event-B Machine for WXR Model

Intentionally removed the Event-B code

A Framework for Explicit Modelling of Domain Knowledge in State-Based Formal Methods: The Case of Interactive Critical Systems

System engineering advocates an explicit modelling of domain knowledge at early stages of the development cycle. Moreover, integrating contextual information and certification standard requirements into formal models enhances their quality and reliability. On the one hand formal methods provide primitives for modelling components and views of these systems but they are not endowed with built-in primitives for explicit modelling contextual constraints, and more broadly, domain knowledge associated with these formal models. Consequently relevant domain knowledge is implicitly hardcoded in the system formal specification or is, in the worse case, overlooked. On the other hand ontologies have demonstrated their efficiency in modelling domain-specific features but they are not available as built-in primitives in formal methods.

The goal of this thesis is to propose a framework and associated methodology for modelling explicitly domain knowledge in formal modelling. As a byproduct, consequences of the explicit modelling of domain knowledge are investigated including formal standard-based conformance checking of interactive critical systems and domain-specific behavioural analyses of formal models. In our research effort, we defined an integrated framework for addressing the explicit modelling of domain knowledge problem in formal modelling. The framework specified three main steps: formalising domain knowledge, annotating formal models and transferring domain knowledge constraints. The formalisation is achieved by proposing an ontology modelling language in the form of a generic Event-B theory, the annotation consists in typing elements of the model with the concepts of an ontology and the transferring of predefined domain properties is supported by the description of a methodological rule of using exclusively the data types and operators of the Event-B theory (closure). Furthermore, several cases studies from the interactive critical systems realm have been addressed to showcase the generality, effectiveness and advantages of the framework. First, conformance checking is a prominent consequence of explicit modelling of domain knowledge; standard conformance to ARINC 661 certification standard of a cockpit application user interface is addressed as a special case of explicit modelling of domain knowledge. A second consequence of the framework is the definition of a method and formal Event-B theories for specifying domain-specific behavioural analyses.

Keywords: Formal Methods, Interactive Critical Systems, Event-B, Domain Knowledge and Ontologies

Un cadre pour la modélisation explicite des connaissances de domaine dans les méthodes formelles orientées états: le cas des systèmes critiques interactifs

L'ingénierie système préconise une modélisation explicite des connaissances de domaine aux premières étapes du cycle de développement. De plus, L'intégration d'informations contextuelles et d'exigences provenant de normes de certification dans des modèles formels améliore leur qualité et leur fiabilité. D'une part, les méthodes formelles fournissent des primitives d'abstraction pour modéliser la structure et les comportements de ces systèmes, mais elles ne sont pas équipées pas de primitives spécifiquement dédiées pour modéliser explicitement les contraintes contextuelles, et plus largement, les connaissances de domaine associées à ces modèles formels. Par conséquent, les connaissances de domaine sont implicitement codées en dur dans la spécification formelle du système où sont, dans le pire des cas, ignorées. D'autre part, les ontologies ont démontré leur efficacité dans la modélisation des connaissances, mais elles ne sont pas fournies en tant que primitives dans les méthodes formelles.

L'objectif de cette thèse est de proposer un cadre et une méthodologie associée pour modéliser explicitement les connaissances de domaine dans le contexte la modélisation formelle. Par ailleurs, les conséquences de la modélisation explicite des connaissances de domaine sont étudiées, y compris la vérification formelle de la conformité par rapport aux normes des systèmes critiques interactifs ainsi que les analyses comportementales spécifiques au domaine appliquées à des modèles formels. Dans notre effort de recherchen, nous avons défini un cadre intégré pour résoudre le problème de la modélisation explicite des connaissances de domaine dans la modélisation formelle. Le cadre a spécifié trois étapes principales, à savoir formaliser les connaissances de domaine, annoter les modèles formels et enfin transférer les propriétés des connaissances de domaine. La formalisation est réalisée en proposant un langage de modélisation à base d'ontologies sous forme de théorie Event-B générique, l'annotation consiste à typer les éléments du modèle avec les concepts d'une ontologie de domaine et le transfert de propriétés de domaine prédéfinies est régi par la description d'une règle méthodologique stipulant l'utilisation exclusive des types de données et des opérateurs de la théorie Event-B formalisant une ontologie de domaine. En outre, plusieurs études de cas fournies dans le domaine des systèmes critiques interactifs sont abordées pour montrer la généralité, l'efficacité et les avantages du cadre. Premièrement, la vérification de conformité est une conséquence importante de la modélisation explicite des connaissances de domaine dans la modélisation formelle ; la conformité à la norme de certification ARINC 661 d'un système interactif d'une application de cockpit est utilisée pour démontrer l'efficacité du cadre. Une deuxième conséquence du cadre est la définition d'une méthode et de théories d'Event-B pour spécifier des analyses comportementales spécifiques à un domaine. En outre, une étude de cas concrète est décrite et analysée pour illustrer la méthode conçue.

Mots-clés: Méthodes Formelles, Systèmes Critiques Interactifs, Event-B, Connaissances de Domaine et Ontologies